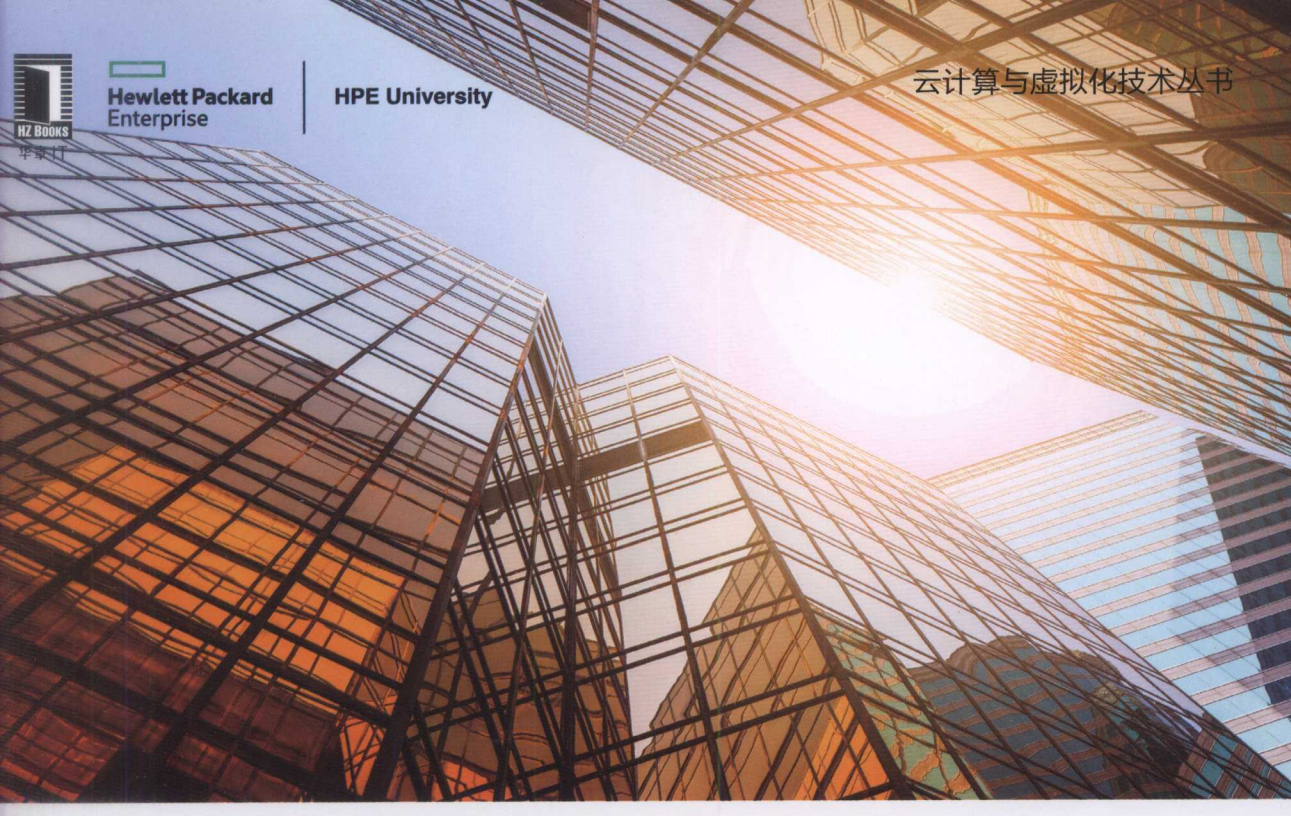


Hewlett Packard Enterprise | HPE University

云计算与虚拟化技术丛书

Architecture Development Guide for Enterprise Level Container Cloud

企业级容器云架构开发指南

HPE ku8s team 编著

涵盖大规模容器集群管理架构、不同分布式应用混合架构系列技术、企业级容器云的应用实践，构建云计算与容器技术的完整技术储备

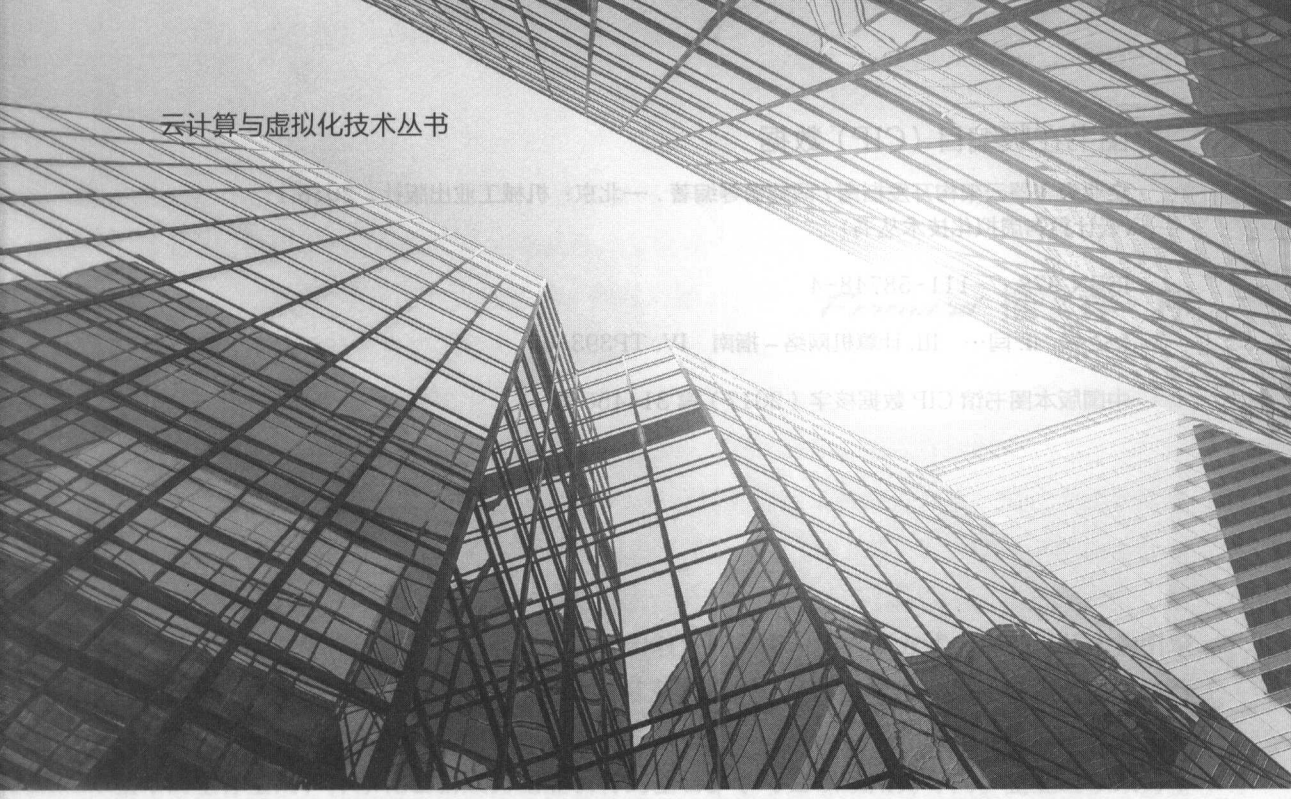
机械工业出版社
China Machine Press

内容简介

这是一个日新月异的IT世界，各种新概念、新技术层出不穷，特别是云计算领域围绕着容器、微服务、DevOps的新兴势力更是如日中天。如何才能快速全面理解这些新技术以及它们之间的关系？为此，慧与大学携手慧与（中国）有限公司企业云计算领域的资深专家，共同编写了此书。在书中，您将对以下的内容有更深刻的认识与理解：

- 云计算技术的发展与现状
- 如何全面理解微服务
- DevOps实践框架
- Docker价值及生态圈
- Kubernets理论及实践
- Mesos快速入门
- 容器云技术的企业实践分享

云计算与虚拟化技术丛书



Architecture Development Guide for
Enterprise Level Container Cloud

企业级容器云架构开发指南

闫健勇 龚 正 吴治辉
屈晓萌 王健飞 王 伟 刘晓红 编著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

企业级容器云架构开发指南 / 闫健勇等编著 . —北京：机械工业出版社，2018.1
（云计算与虚拟化技术丛书）

ISBN 978-7-111-58748-4

I. 企… II. 闫… III. 计算机网络 – 指南 IV. TP393-62

中国版本图书馆 CIP 数据核字（2017）第 314450 号

企业级容器云架构开发指南

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：佘 洁 责任校对：殷 虹
印 刷：北京市荣盛彩色印刷有限公司 版 次：2018 年 1 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：15
书 号：ISBN 978-7-111-58748-4 定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

Foreword 推 荐 序

近年来，数字体验与万物互联的融合开始触发连接用户的崭新方式，各种以技术为引擎的创新业务增长模式蜂拥出现，使得 IT 不但支撑业务，而且深度融入到业务模式当中进行生产和创新。在这个差异化和定制化的时代，仅仅学习、拷贝业界先进技术已经不足以保障基本的生存空间，体系化创新成为基业长青的基石。学习必以致用为导向。企业应该致力于为客户提供精益而卓越的服务，做好客户的千里眼、顺风耳，让数字化生活触手可及。创新，当始终以此为愿景，引入适合企业自身的新技术，整合数字化企业核心系统，持续培养关键技术人才，进行精益企业思维与敏捷技术的转变。

在我所供职多年的企业里，云计算技术应用已经经历了从概念学习、力行实践，到使能创新的三个阶段。目前 IT 基础设施已经全面云化，企业级 PaaS 平台初具规模，基于平台模式构建的生态体系助力企业和合作伙伴的融合创新。在云化的进程中，容器云技术实践使企业的云化能力实现提速，云计算已经成为重要的使能者，让传统电信级业务的全面云化成为可能。

虚拟化将 IT 基础设施从计划供给转换成了像自来水一样的按需供给，为 IT 产业的精益生产落地提供了基础。微服务用传统中国分而治之的思想解决了现有大型系统大而重的问题，使敏捷和灵动变为可能。而 DevOps 更是采用自动化一切方针，依托工具将原有 IT 手工作坊跃升成了高可用的生产流水线。这三者的相辅相成，汇成了一个词：效率。降本增效，是所有 IT 运营支撑的关键词。

在运营支撑系统经历了初创到壮大，到面临转型的时间点迎来了这本书和容器云课程，本书的作者都是十几年来始终活跃在一线的咨询和研发精英，他们拥有国际化视野，怀揣着对新技术的热爱，并致力于实施落地过程中的创新，将奋战在大型 IT 系统多年的实践经验与理论结合，讲述了理想与现实，描绘了挫折与奋进。

刘 虹

中国移动通信集团公司业务支撑系统部副总经理

前　言 *Preface*

美国国家标准与技术研究所（NIST）对云计算有如下定义：云计算是一种允许用户通过网络便捷地接入可动态配置的共享资源池（包括网络、存储、计算能力、应用以及业务），并以最小的管理成本实现对这些可配置计算资源的快速申请、使用和释放的技术架构和服务模式。作为新一代的技术架构和服务模式，云计算改变了很多行业的现状，大量企业可以像使用水电等资源一样来使用IT资源。云计算提高了IT系统的扩展性，许多企业现在愿意将云平台作为他们的IT基础设施，云计算模式正在成为标准，成为一场改变人们生活方式和企业经营方式的革命。

云计算从最初的概念出现到如今的应用普及，已有十余年的时间。相比云计算诞生初期的情况，目前的技术条件和行业环境已经发生了巨大的变化，云计算开始进入大规模应用阶段，显著改变着我们生活的各个方面，而产生当前效应的一个重要因素就是容器技术的大规模应用。

2013年年初，dotCloud公司将内部项目Docker开源，之后Docker这个名字迅速蔓延到整个信息产业。容器技术并不是全新的概念，Docker所采用的关键技术也早已存在，但正是由于Docker的出现，使得以容器技术来构建云计算平台变得更加方便和快捷。容器技术不仅改变了系统架构的设计方式，还改变了研发过程和系统运维的方式，使得我们一直期望的开发速度更快、系统质量更好、更易维护的IT系统变成现实。Docker的出现是云计算发展的重要里程碑，成为云应用大规模推广的基石。

本书由慧与大学组织，由慧与（中国）有限公司容器云项目组的人员完成，书籍内容的蓝本为慧与大学颇受学员欢迎的容器云系列课程的课件。参与本书编写的诸位作者常年活跃在技术一线，同时作为慧与大学的讲师和技术顾问，为大量客户提供了专业的培训服

务，并且将他们在技术一线的经验传递给慧与大学的学员。慧与大学为了能够让更广泛的人员受益，组织了本书的编写。本书从技术实践者的角度出发，介绍了当前主流的 Docker、Kubernetes、Mesos 等容器和管理技术，同时对微服务架构设计与实现、DevOps 开发与交付的方法和实践进行了介绍。书中既有对技术概念、流程和方法的介绍，也有对具体技术实现和案例场景的介绍，我们期望通过这样的内容设计，帮助读者在全视角了解云计算和容器技术的情况下，能将涉及的相关技术和方法直接应用到工作中，解决实际问题。

全书分为五个部分。第一部分（第 1 章）对云计算进行了概要性介绍，使读者建立对虚拟化、容器技术、公有云和私有云的基本概念。第二部分（第 2 章）对微服务架构的设计和实现进行了介绍。第三部分（第 3 章）关注研发生产力，介绍了 DevOps 的概念和实践。第四部分（第 4 ～ 6 章）对 Docker、Kubernetes、Mesos 分别进行了介绍。第五部分（第 7 章）介绍了企业级容器云在电信行业的应用实践，使读者对从容器技术到大规模容器集群管理架构，到不同分布式应用混合架构这一系列技术，再到企业级容器云的应用实践均能够有所了解。本书的五个部分既彼此独立，又相互关联，能够帮助读者建立起云计算和容器技术的完整技术储备。

本书适用于系统架构师、开发和测试人员、运维人员、企业 IT 主管、系统管理员，也适合作为高等院校计算机专业学习云及容器技术的教材。

凌云意气，有容乃大。在信息技术快速发展的今天，希望本书能为读者带来实际的收益。慧与大学愿意和读者一起积极投身到容器云产业的实践中，促进信息产业的蓬勃发展。

慧与大学

目　录 *Contents*

第 1 章 Chapter 1

云计算概述

1.1 虚拟化技术简史

1.1.1 虚拟化技术的起源

虚拟化（virtualization）技术总体上是软硬件相互协作而产生和发展的技术，可以将其视作服务器的“多路复用”技术，即将一台物理服务器模拟（分割）成为多台逻辑上相互独立的服务器——虚拟机。虚拟化技术确保了运行于一台物理机上的多个虚拟机之间具有严格的隔离机制，彼此不会相互影响。对于一个高配置的物理机，通常可以虚拟化为几十台虚拟机，每个虚拟机可以安装不同的操作系统，以供不同的租户选择使用，而在每个虚拟机的租户看来，他们完全“独占”该物理机。

在计算机发展的初期，大型机非常昂贵，也经常得不到充分利用，采用虚拟化技术，将一个大型机虚拟化成几十个虚拟机供其他部门使用，可提升资源的使用效率。因此虚拟化技术是计算机发展的必然结果。

大型机的虚拟分区技术最早可以追溯到 20 世纪六七十年代。早在 1965 年，IBM 公司就发明了虚拟化技术，允许用户在一台 IBM 的大型主机上运行多个操作系统，从而让用户尽可能地充分利用昂贵的大型机资源。最早使用虚拟化技术的产品是 IBM 7044 主机，而“虚拟机”这个专用术语也是首次被 IBM 7044 主机所正式采纳。到了 1965 年，IBM System/360 Model 67 成为真正意义上的虚拟机的鼻祖，它通过 VMM（Virtual Machine Monitor）对所有的硬件接口都进行了虚拟化。而后来的 IBM VM/370 则更进一步，首次实现了虚拟化嵌套能力，即在一个虚拟机上继续虚拟出多个虚拟机，而在 X86 服务器上这项技术直到 2000 年左右才获得了突破。

虽然早在20世纪60年代就已经出现了成熟的虚拟机产品，但直到1977年IBM的管理层才完全接受虚拟机这个强大的新生事物，而仅在一年后（1978年）IBM就已经在全球创建了1 000多个虚拟机了。之后虚拟化技术快速发展，以IBM System z系列为代表的大型机虚拟化技术逐步渗入UNIX小型机的领域，成为小型机的标配技术之一。

在IBM之后，惠普也在自己的UNIX服务器上提供了虚拟化技术，并提供了多种虚拟化方案，每种方案都有针对的需求场景，因此惠普逐渐成为小型机（后文简称小机）虚拟化领域的实力厂商。如图1-1所示，惠普的虚拟化方案总体分为以下几种：

- 硬件分区。
- 虚拟分区（软分区）。
- 安全资源分区。

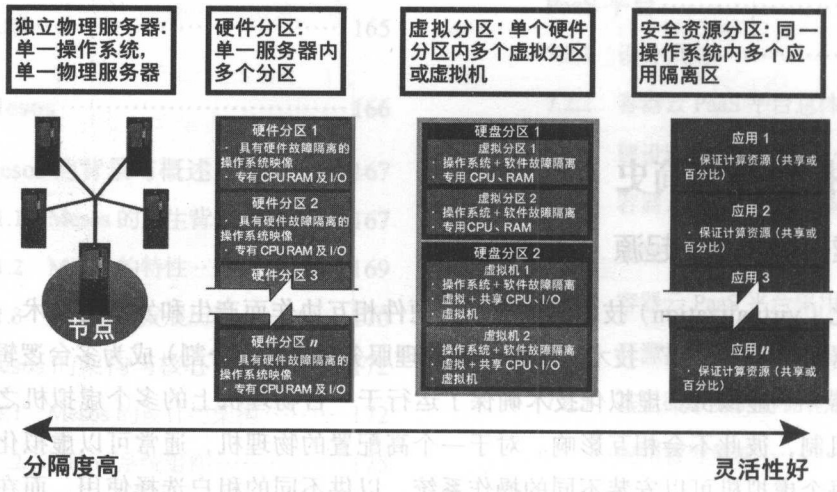


图1-1 惠普的虚拟化方案

其中，硬件分区是指一个单一物理服务器分割为几个不同的虚拟机；虚拟分区则是进一步在一个硬件分区里继续划分多个虚拟分区（或多个虚拟机）；而安全资源分区则是在一个虚拟机内划分多个应用隔离区，类似后来的容器技术。

这3种分区技术的应用场景有所区别：硬件分区适合于性能与隔离程度要求高的虚拟化场景，在这种情况下，不同分区内的物理资源完全独立，CPU、内存、I/O等都完全隔离；虚拟分区是从软件层面进行的资源分区，由于同一个物理分区下的多个虚拟分区共享同一份底层物理资源（CPU、内存、I/O等），因此资源的隔离性与性能会差一点，但其优点是分区很灵活，可以随心所欲地设计虚拟分区的大小和数量，多用于测试性能要求不高的虚拟化场景；安全资源分区主要是在安全的场景下进行分区，类似后来的Docker容器技术，具备快速实施以及所占资源少的优点，是一种轻量级的虚拟化技术。

1.1.2　X86 平台虚拟化历史

虚拟化技术在大型机时代诞生，在随后的小机时代得到进一步发展，而虚拟化技术的真正爆发则是在“X86 时代”，图 1-2 是 X86 平台虚拟化进程的示意图。1999 年虚拟化技术首次亮相 X86 平台，随后加速普及到个人计算机，正所谓“旧时王谢堂前燕，飞入寻常百姓家”。

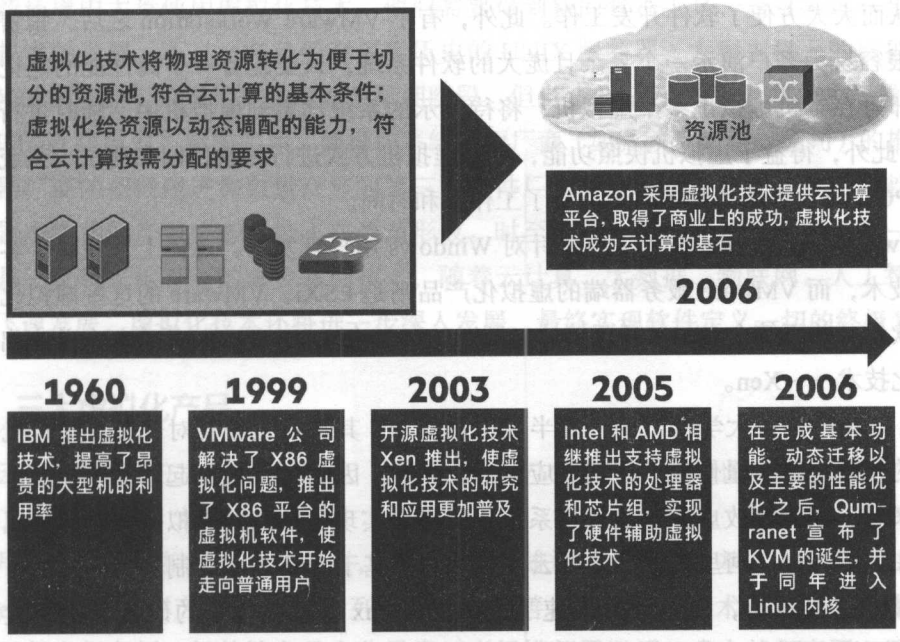


图 1-2　X86 平台虚拟化进程

长期以来，由于 X86 CPU 缺乏虚拟化技术所需的关键指令，不能很好地支持虚拟化技术，导致上层相关的虚拟化软件很难实现。后来 Intel 与 AMD 意识到虚拟化的重要性，在其新一代的 CPU 上提供了专门用于虚拟化的指令集，从而加速了 X86 虚拟化的发展。此外，X86 处理器的性能长期以来大大落后于 UNIX 小机的处理器，因此虚拟化的价值并不大，但随着 X86 处理器的加速多核化发展，其性能也得到了飞跃式提升。随着硅谷互联网泡沫的破灭，UNIX 服务器的市场份额快速萎缩，被 X86 服务器侵蚀了大部分市场。后来在云计算浪潮的推动下，X86 虚拟化技术才首次被真正重视起来。我们看到，X86 虚拟化具备了天时、地利、人和的优势，所以注定爆发。

1999 年是 X86 虚拟化技术发展的一个重要里程碑：这一年，VMware 公司首次解决了 X86 平台的虚拟化问题，推出了面向 X86 平台的商业化虚拟机软件——VMware Workstation，使虚拟化技术开始走向普通用户。

VMware 公司的 Workstation 虚拟机软件可以让普通用户在自己的 Windows 笔记本式计算机或者台式计算机上很方便地创建出一个虚拟机，并且可以在这个虚拟机上安装各种版本的 Linux 系统或者 Windows 系统。我们知道，企业级应用大部分都是安装和部署在 Linux 服务器上的，为了开发并测试这些应用，项目组通常需要一个专门的服务器，有了 VMware Workstation 之后，一个普通的笔记本式计算机就可以创建出几个 Linux 虚拟机来，在无须其他 IT 人员介入的情况下，开发和测试人员就可以独立完成整个系统的部署和测试过程，从而大大方便了软件开发工作。此外，有了 VMware Workstation 之后，销售或者售前团队很容易为客户演示一个复杂且庞大的软件系统，只要配备一个高性能的笔记本式计算机，即可在本机虚拟出多个虚拟机，将待演示的系统部署到虚拟机中，随时进行“移动演示”。此外，得益于虚拟机快照功能，采用虚拟机方式进行的软件回归测试、系统备份恢复以及 POC 演示等活动都极大地节省了工作量和时间。

VMware Workstation 技术主要是针对 Windows 用户开发的，总体上来说属于客户端的虚拟化技术，而 VMware 服务器端的虚拟化产品则是 ESXi。VMware 的这些虚拟化产品都是商业软件，并不符合 Linux 开源精神。所以，2003 年 Linux（X86 平台）上首次出现了开源虚拟化技术——Xen。

Xen 是英国剑桥大学推出的一种半虚拟化技术，其特点是需要对部署在虚拟机上的操作系统的代码进行少量修改，使其适应虚拟化环境，因此 Xen 使用起来有些麻烦。但从另一方面来看，因为修改虚拟机的操作系统代码可以实现比较高的虚拟化性能，所以目前公有云巨头如亚马逊、阿里巴巴的公有云虚拟机都是基于 Xen 技术定制演变而来的。虽然目前 Xen 技术已经没落，但它大大加速了 X86 与 Linux 虚拟化技术的研究与应用过程，可以说功不可没。但 Xen 需要修改虚拟机上的操作系统才能实现虚拟化，因此也存在种种弊端和限制，如第三方修改 Windows 这种闭源操作系统的代码是违反软件许可证的做法。为了更好地支持全虚拟化，2005 年 Intel 和 AMD 相继推出支持虚拟化技术的全新 CPU 和芯片组，从而在硬件基础上第一次有了重大突破，实现了硬件辅助虚拟化技术。其原理是在 CPU 中加入虚拟化指令，这种指令的加入大大加速了整个 X86 的服务器端虚拟化的发展进程，也促进了新一代 Linux 虚拟化软件的诞生，这就是后起之秀——KVM。

2006 年以色列公司 Qumranet 宣布了新一代 Linux 虚拟化软件 KVM 的诞生，KVM 在不到一年的时间里即被纳入 Linux 内核中并得到支持，成为第一个 Linux 内核级支持的开源虚拟化软件。KVM 实际上已经成为云计算标配的虚拟化技术，也就是从 KVM 被推出之后，大规模的云计算开始在各个行业当中得到推广，OpenStack 底层的虚拟化技术即采用了 KVM。

总体来说，X86 平台的虚拟化从技术上来讲，主要分为半虚拟化和全虚拟化两种。Xen 属于半虚拟化技术，需要对客户操作系统进行定制化改造，这样才能在虚拟层上运行，半

虚拟化为用户提供了一个可以改造的通道，可以对虚拟机操作系统进行定制化的改造；后来的KVM则属于全虚拟化，即无须对操作系统进行任何改造，直接就可以在虚拟机上安装。Xen需要对操作系统进行一定的改造，而且长期以来没有被纳入Linux内核中，慢慢地被边缘化了，而KVM则成为真正的赢家。

因为X86平台可以提供便宜、高性能和高可用的服务器，所以X86平台的虚拟化技术突飞猛进，并且首次向人们展示了虚拟化应用的广阔前景。更重要的是，一些用户已经开始在生产环境中大量使用虚拟化技术，他们需要得到新的管理工具，反过来进一步促进了虚拟化技术的发展。不过，与已经有多年历史的UNIX服务器、大型主机上的虚拟化技术相比，X86服务器上的虚拟化仍旧处于早期阶段，但自2006年以来，从X86处理器层面的AMD和Intel积极发布新的芯片，到操作系统层面厂商（如微软、红帽等公司）的推动，以及服务器厂商的积极跟进和数量众多的第三方软件厂商不断涌现，我们看到一个趋于完整的服务器虚拟化的产业生态系统正在逐渐形成。时至今日，虚拟化技术已经不是一个新技术，而是一个越来越普遍的重要基础技术，随着云计算、大数据、物联网、人工智能等新技术的飞速发展，虚拟化技术还将进一步深入发展，最终实现软件定义一切的终极梦想。

1.1.3 三大虚拟化产品

在云计算领域有三大虚拟化产品，它们分别是VMware公司推出的ESXi商业产品以及开源的Xen和KVM，如图1-3所示。

最早一批尝试虚拟化技术的公司，特别是实力雄厚的大公司，基本上都购买并使用过VMware的ESXi系列产品。ESXi系列产品十分完善，其虚拟化技术是独有的，性能也不错，使用起来很方便。但其缺点是十分昂贵，当规模部署以后，成本就很高了。因此一般公司实施小规模云计算的时候，都会以VMware产品为起点，但随着集群规模的扩大，慢慢地都会迁移到开源的Xen或者KVM之上。

Xen是一个开放源代码的虚拟机监视器，最早由剑桥大学开发完成，Xen允许在单个计算机上运行多达100个虚拟机，虚拟机上的操作系统必须进行显式的修改（移植）才能在Xen上运行，虽然修改过操作系统的代码，但操作系统仍然保持用户应用的兼容性。通过修改操作系统代码，使得Xen无须特殊硬件和新的CPU支持就能达到高性能的虚拟化。Xen开通了一个通道，允许虚拟机操作系统跨过虚拟化层直接操作底层硬件，这能保证虚拟机的性能基本达到裸机运行的性能，因此前期诸如亚马逊、阿里巴巴等公有云的提供者基本上都会在这个基础上做一些自己的定制化，以此来提供高性能的公有云服务。Xen在早期不被Linux内核支持，要实现Hypervisor功能，IT管理员需要把开源Xen作为主流内核的补丁来安装，甚至安装后不能对内核进行升级，否则会破坏Xen的功能。难与Linux内核集成的这个缺陷导致后来者KVM一出现就备受关注。

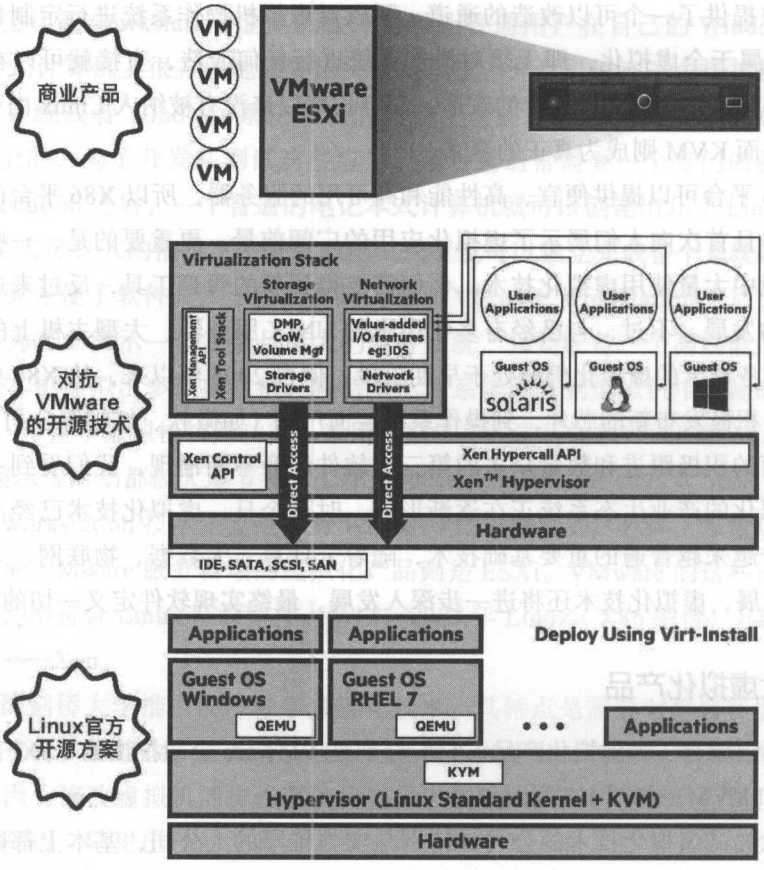


图 1-3 三大虚拟化产品

与 Xen 半虚拟化不同的是，KVM 是全虚拟化技术，直接在 Linux 内核运行，KVM 创建的虚拟机直接可以安装 Windows、Linux 等系统，使用起来也相对方便，但其缺点是由于存在虚拟化的硬件层，所以其网络 I/O 和存储 I/O 的吞吐性能会受到影响。虽然可以通过一定的技术降低这个影响，但总体来说虚拟化还是会对 I/O 吞吐性能产生一定的影响，在做系统设计的时候要充分考虑这个问题。

前有 VMware，后有 KVM，为了能在虚拟化的地盘上继续分一杯羹，Xen 也不断努力，经过漫长的改进，终于在 2011 年年初获得了 Linux 的完全支持，但已经错过了提高市场占有率的绝佳机会。而且就在 Xen 努力靠近 Linux 内核的这些年，KVM 已经获得了长足的发展，之前支持 Xen 的厂商也改变风向。例如：红帽很快就收购了 KVM 技术，不断完善自家产品，Red Hat Enterprise Linux（RHEL）5 采用的还是 Xen Hypervisor，但在 RHEL 6 中就移除了所有 Xen 相关组件，只用 KVM，并且提供了 Xen 到 KVM 的虚拟机迁移工具；另一家 Linux 厂商 Ubuntu 则明确表示选择 KVM 作为其 Hypervisor。除了红帽

和 Ubuntu，KVM 也受到了 IBM 的关注，IBM KVM（北京）卓越中心落户北京，帮助中国客户、软件开发商及合作伙伴更好地采用开放的虚拟化技术，以支持其云计算项目的发展。此外，KVM 作为默认的开源软件 Hypervisor，获得了开源虚拟化联盟（OVA）的认可。由于大厂商和众多团体机构都在推广 KVM，这使得 Xen 在开源虚拟化市场上难以立足，Xen 这个成熟的虚拟化平台昔日在可用资源、可管理性、性能等方面的优势在快速发展的 KVM 面前也略显暗淡，随着时间的推移，慢慢地无人再提及，可能在不久的将来，Xen 将被人遗忘。如果坚持在 Linux 平台上使用开源 Xen，SUSE Linux Enterprise Server 和 Oracle 将是唯一的选择，但 Oracle VM（来自 SUN 的产品）这个基于 Xen 的虚拟化平台很难直接用于 Oracle 现有的用户群，Oracle 公司在企业 Linux 领域只是小玩家，对 Xen 的支持也很有限。

1.1.4 私有云与公有云

在私有云的建设当中，除了 VMware 可以提供商业化的产品，目前比较流行的是基于 OpenStack 开源项目搭建自己的私有云。OpenStack 本身并不是一个技术，实际上是一个产品体系，包含很多的模块和功能。例如，主机的虚拟化模块 Nova 用来创建 KVM 虚拟机；Neutron 模块实现了网络虚拟化的功能；Swift 是与存储相关的模块；Keystone 模块是 OpenStack 平台的认证和鉴权模块；Horizon 提供了 Web 界面来方便用户使用，是整个平台的管理工具；Ironic 项目则希望 OpenStack 把裸机的管理也纳入进来，被纳入的裸机上面不运行虚拟机，Ironic 的发展是为了适应目前虚拟化的新趋势，即越来越多的基于 Docker 的应用直接运行在裸机上，而不是在虚拟机上再进行容器化，因为后者的性能损耗比较多；另外一个新的与容器相关的模块最近也被纳入 OpenStack 中，即谷歌在背后支持的 Magnum 项目，它的目标是让 Docker 容器以及谷歌的 Kubernetes 架构成为 OpenStack 上的“一等公民”，被 OpenStack 完美支持，Magnum 项目可以被视作容器技术首次与虚拟机技术平起平坐的一次重要尝试。虽然 OpenStack 仍然在不断改进和发展，也在尝试拥抱新一代的容器技术，但不可否认的是曾经的巨无霸 OpenStack 正在慢慢地没落。

在公有云方面，目前比较流行的几个公有云如微软的 Azure、阿里云、Rackspace、Amazon Web Services、Gooole Compute Engine 等都对外提供虚拟机，企业和个人可以在网上购买虚拟机，订购虚拟机的时候，用户可以选择任意规格的型号，如几个 CPU、内存多大、硬盘多大，下单付款之后，可以立即在线开通虚拟机，这实际上就是我们比较熟悉的 IaaS（Infrastructure as a Service）平台。公有云的最大好处就是提高了资源交付的能力和速度，加快了互联网应用的开发和上线速度，而这背后的最大功臣就是虚拟化技术。

1.2 虚拟化热点技术与终极目标

1.2.1 网络虚拟化

虚拟化技术体系的核心是计算虚拟化，体现为一个物理主机分割为多个虚拟机，但整个虚拟化技术体系中最难的则是网络虚拟化技术，这对绝大多数软件工程师和IT人员来说，都是一个高门槛的领域。

互联网已经成为我们生活中必不可少的一部分，可能对于很多人来说，一年的语音通话时长要远远少于上网和网络即时通信所花费的时间。对于一台无法联网的计算机来说，它的价值甚至不如一个能上网的手机。对于虚拟机也是一样的，如果虚拟出来的虚拟机不能与其他虚拟机进行通信，不能访问外网或不能被外网访问，那么虚拟机的价值将会大打折扣，公有云将不会存在。

怎么理解网络虚拟化？简单来说，原来是一个物理服务器作为单独的通信端点存在，但虚拟化后，变成多个拥有独立IP的虚拟机。这些虚拟机跟物理机一样，也需要挂接到交换机上进行通信，虚拟机的IP地址与物理机的IP地址属于两个不同的IP地址平面，相互是隔离的，如果一个物理机上的虚拟机与另外一个物理机上的虚拟机要进行通信，那么通信的报文就必须通过物理机的IP地址和物理链路进行传递，所以就出现了所谓的报文封装的概念和技术，如GRE隧道或者VXLAN以及其他厂商的私有方案。

通常来说，位于一个机房或数据中心的所有物理机属于同一个局域网，延伸开来，这些物理机上的所有虚拟机也属于同一个虚拟局域网。由于虚拟机的数量可能是物理机数量的10倍甚至100倍，所以这个虚拟局域网又被称为“大二层”交换网，即总体上是一个扁平的二层网络，只有交换而没有路由。我们知道，网络通信严重依赖于网络设备，如网卡、交换机、路由器，与CPU不同的网络相关规范标准与硬件升级都是一个很缓慢的过程，所以网络虚拟化发展缓慢，目前主要还是采用软件模拟的方式来兼容已有的网络协议和网络设备，导致虚拟化网络的性能成为明显的瓶颈。但可以预见的是，不久的将来，越来越多的网络设备将会直接支持网络虚拟化技术，如VXLAN与GRE。

1. 网络虚拟化之SDN

除了“大二层”的底层网络虚拟化技术之外，网络虚拟化目前在朝着两个比较热门的高层领域加速发展。第一个是SDN（Software Defined Network，软件定义的网络）。SDN实际上是想抛弃传统的网络管理模式，通过虚拟化平台提供的网络管理界面，采用软件的方式来完成任意业务所需的底层网络的规划、部署、扩容和自动化管理能力，即通过简单的软件操作就能完全控制网络的定义和管理。SDN希望虚拟机所依附的网络也像虚拟机那样能够实现灵活定义、动态改变和自动化管理。

SDN 之所以会兴起，背后有深刻的原因：传统网络的规划建设很多时候没有办法覆盖可能承载的各类业务，而且一旦建成，就基本上一成不变，但业务则是不断发展和变化的，底层网络的不变与其上业务的不断变化就产生了矛盾。此外，在这个竞争加剧的互联网时代，软件敏捷交付以支持业务快速上线的能力正变得越来越重要，虚拟化技术的不断发展也进一步促进了这方面的进步。但网络这部分则始终游离在软件交付的控制范围之外，如果待交付的软件需要对网络做一些定制的配置，如开启某些端口、配置某些路由，则软件团队必须要提交一个工单给网络管理员（或者网运部门），后者经过漫长的审批和施工才能完成工单。如果在这个过程中信息有所缺失或者沟通不畅，那么等待的时间将会大幅增加。在这个场景中，我们看到底层网络与上层应用的第二个越来越突出的矛盾：网络与应用完全隔离，缺乏联动。因此，SDN 技术必然出现，可以将其看作以软件为核心的虚拟化技术进一步蚕食传统网络供应商地盘的结果，SDN 要淘汰原先与应用无关的传统网络管理和运维模式，使得软件敏捷交付变得更加敏捷，进一步提升了软件技术在整个 IT 行业的领导力和价值。

SDN 把网络这一层拆分为三部分：第一部分是数据平面（Data Plane），专门负责数据传输，相当于网络底层的传输部分；第二部分是控制平面（Control Plane），它属于 SDN 的核心和主要模块，负责对网络设备下发管理指令，所有网络设备都由集中的一个 Controller 控制，底层的协议是标准的，纳入 SDN 系统的所有设备都要遵循标准协议，因此 Controller 可以无缝对接和管理不同厂家的数据平面网络设备，成为实际上标准化的大网管平台；第三部分就是 SDN Controller 定义的网络管理接口，这些接口采用了流行的 REST API 方式进行定义，这样一来，上层应用就可以采用 REST API 来实现 SDN 的宣传理念——软件定义网络，如编程方式定义路由策略、流量控制策略，修改防火墙规则，映射 NAT 端口，创建子网等。SDN 的提出和发展也进一步促进了云计算的大规模推广。但是目前 SDN 碰到了一个比较棘手的问题，即 Controller 对下面的数据平面有标准接口，但是 Controller 开放出来的 RESTful API 没有一个公开认可的标准，所以这一块的接口比较杂乱，每个厂家的接口都不一样，因此上层应用需要针对不同厂家的 SDN Controller 的特点进行相应的适配，这增加了开发的工作量和难度。

2. 网络虚拟化之 NFV

除了 SDN，网络虚拟化的另外一个重点发展方向就是 NFV（Network Function Virtualization 网络功能虚拟化）技术，NFV 主要针对电信行业，背后的推动力量也是电信行业与相关的供应商，图 1-4 为我们展示了 NFV 与 SDN 的关系。

在传统电信领域中的网元设备基本上都是各个专业厂商生产的，这些网元设备的软件与硬件相对都很封闭，大部分采用了专有的硬件，因此成本相对较高，而对于其性能我们也无法很客观地进行评价，专有系统也不利于大规模自动化运维。随着云计算的规模化发

展，电信行业希望将网元设备标准化，底层的网络功能完全采用标准化的 X86 设备支撑，从而搭上了网络虚拟化的顺风车。

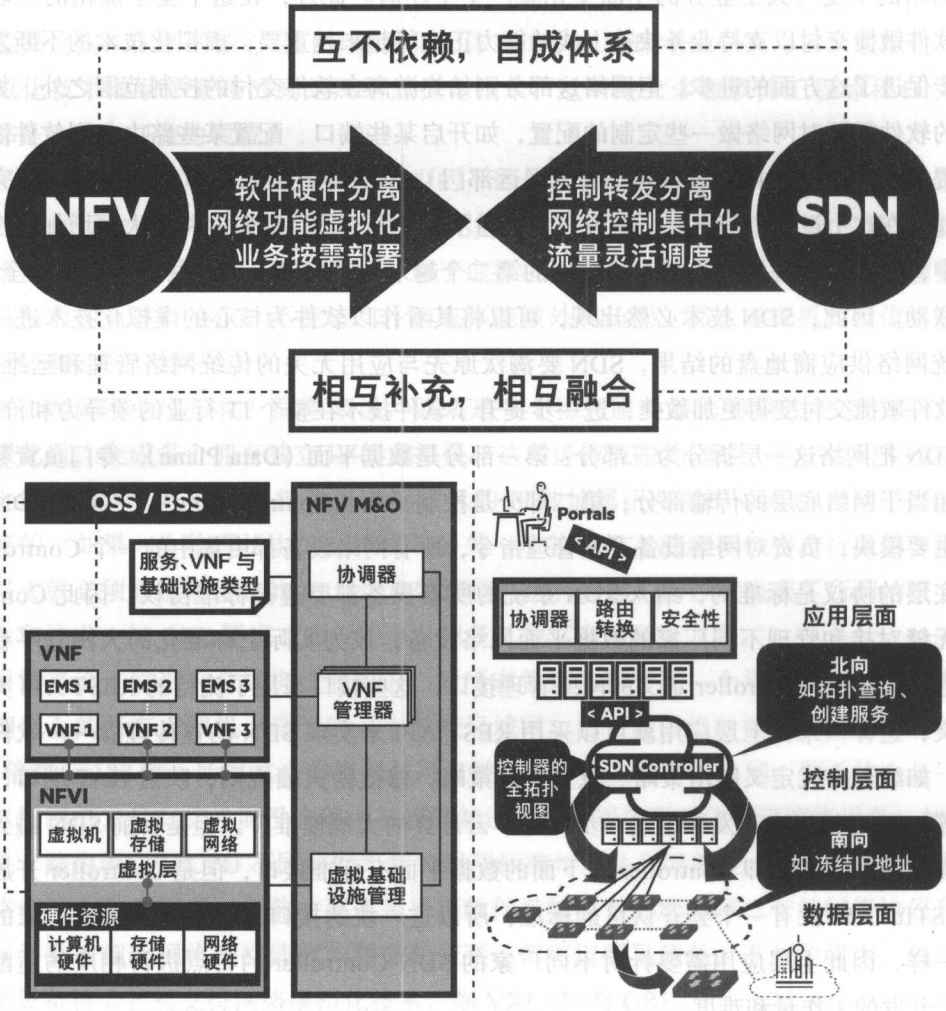


图 1-4　网络虚拟化之 NFV 和 SDN

NFV 将网元底层硬件交给 X86，并且将底层网络功能交给网络虚拟化技术实现以后，网元中的上层电信应用则可以统一在 X86 平台上提供，各个厂商的软件只要符合 NFV 标准规范即可，从而实现了网元设备软硬件的彻底分离及标准化目标。未来各个电信软件开发商只需要提供具有标准化功能的网元软件（程序代码），这些网元软件既可以安装在标准的 X86 通用服务器上，也可以部署在 X86 虚拟机上，这些虚拟机可能是电信运营商内部的资源，也可能是公有云上的资源，因此电信业务很容易根据用户规模的变化而快速弹性伸

缩，并且电信新业务的试点和开展也变得更加容易。正因为前景一片光明，NFV 已成为目前电信行业的一个重点技术，并已在国外很多运营商中得到了大规模的推广，如韩国、美国及欧洲的电信行业中都有相应的 NFV 商用案例。目前国内的中国移动、中国联通、中国电信都在进行 NFV 试点，摸索和积累整个网络向 NFV 迁移的问题和经验。根据规划，2017 ～ 2018 年将是 NFV 大规模地推广或者至少是生产过亿的一年。

1.2.2 存储虚拟化

云计算虚拟化的另外一个重点方向是存储虚拟化，即 VMware 率先提出的“软件定义存储”，如图 1-5 所示。之所以存储设备也会被纳入虚拟化阵营，有两个重要的原因：首先，虚拟机需要硬盘存储，但一台物理机上的硬盘是远远不够数量众多的虚拟机来瓜分的，所以需要将外部大容量存储设备纳入进来；其次，物理机可能发生永久损坏，在这种情况下，虚拟机的文件和数据需要存储在可靠的外部存储上，以便故障后快速恢复。

存储设备实际上是最缺乏标准化支持的，目前的存储产品都是不同厂商根据市场需求进行研发的，相对于网络设备来说，更加专有化和封闭化。不同的场景里有不同的存储虚拟化技术，在选择存储虚拟化方案时，需要综合考虑成本和性能。

例如，在成本不是问题的情况下，可以采用基于光纤的 SAN 存储设备——FC-SAN。其特点是性能特别高，网络存储速度快，一般用在高性能 I/O 的场景，如 Oracle 用 SAN 存储数据库文件；但 SAN 的缺点是端口数量有限，交换机不能任意扩展，所以不适合较大规模的云计算领域。此时可以采用 IP-SAN 存储设备，它基于 IP 的 SAN 技术，其优势是可以在大规模的云计算当中部署并使用；但是它的性能比 FC-SAN 要差一个档次，大概有 10% ～ 15% 的性能损耗，但也基本上能满足一般企业的存储需求，对于 Hadoop 这样的存储要求来说，IP-SAN 绰绰有余。

之前的 SAN 设备实际上提供了块存储能力，除了块存储之外，我们经常使用的还有文件共享存储。这种系统通常由多个 X86 服务器或者存储设备组成一个超大的存储网，并且在其上提供了分布式文件系统，典型的如开源的 GlusterFS、Ceph 等，我们所熟悉的云盘，如百度云盘、360 云盘等，其上都是这类文件的共享存储模式。文件共享存储的特点就是可以建得很大，但是它的读写性能比较差，I/O 存储比较慢，尤其是小数据量的读取，因此适合用作文件备份或者是文件存储，但不适合存储数据库的数据文件。

存储虚拟化的思路与 NFV 的设计理念相似，即把原来各个厂商闭塞的存储设备拆解成真正的存储单元和相关控制单元，并且提供标准的 REST API 接口网关。目前企业级存储虚拟化的思路基本上以 IP-SAN 为核心。其最下层是一个存储池（storage pool），可集成各个厂家的存储设备，如 IBM、惠普、EMC 的存储。其最上层有一个标准的网关，负责控制存储池里的所有存储单元，而在网关之上会开放一些 REST API 接口，允许上层的应用调用。

例如，以软件编程的方式动态定义逻辑卷，并自动挂接存储卷到指定的目标服务器。

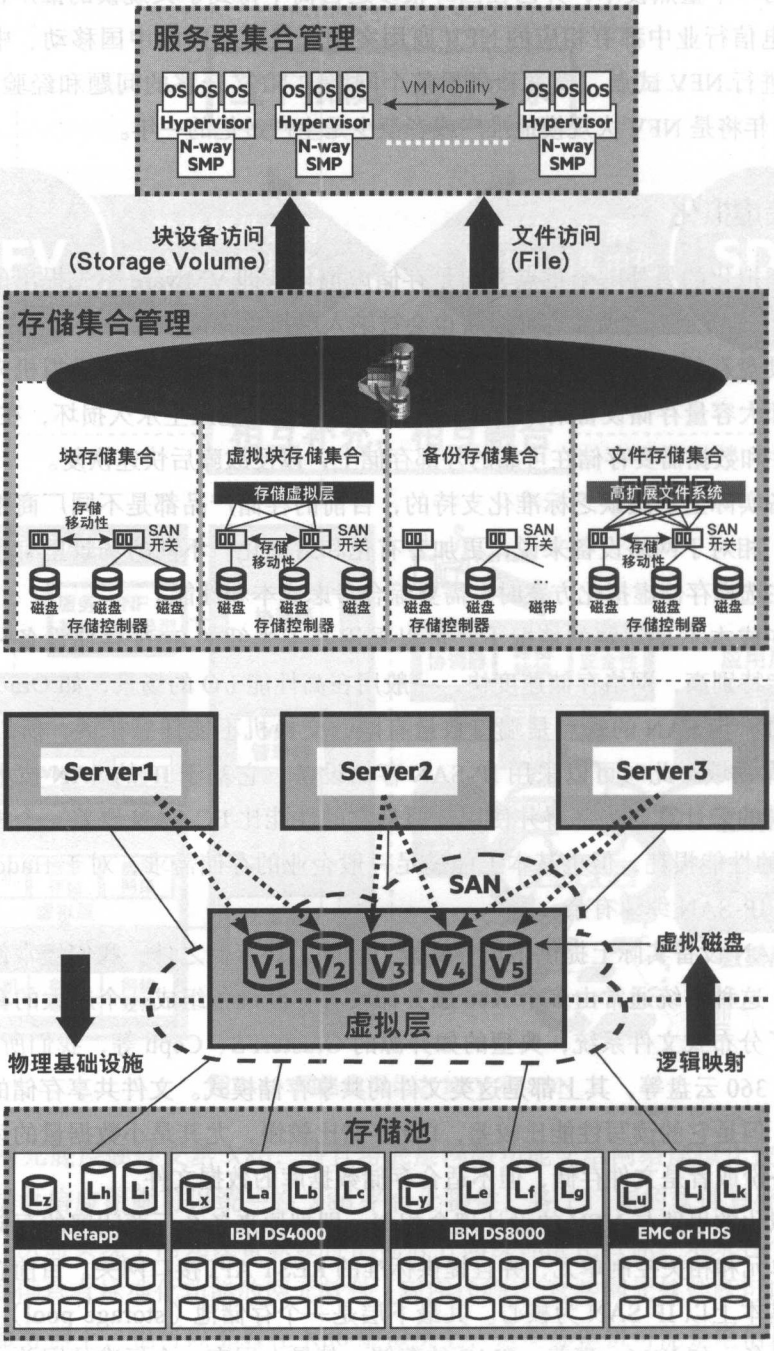


图 1-5 存储虚拟化

1.2.3 虚拟化的终极目标

虚拟化技术可以通过软件的方式在任何时间将一台物理机划分为多台虚拟机来使用，从而大大提高了IT硬件资源的利用率，使得IT资源能够真正成为一项社会基础设施。我们已经看到，建立在虚拟机技术基础之上的以亚马逊为首的一批公有云服务商正在加速影响我们所熟悉的IT生态区，越来越多的应用“跑”在公有云的虚拟机上，服务于各行各业。此外，以虚拟化技术为关键核心的IaaS平台则进一步强化了我们对IT基础设施资源的灵活调度、按需分配、弹性扩展、跨域共享、容灾备份等高级管理能力。

那么，虚拟化的终极目标是什么？

从主机虚拟化到网络虚拟化再到存储虚拟化，我们发现所有的IT基础设施和资源都已经全盘虚拟化，在虚拟化的基础上再进一步实现运维的自动化、管理的标准化，就实现了整个数据中心的虚拟化目标，打造了虚拟化技术的终极目标：实现一个“软件定义”的纯虚拟化的数据中心，如图1-6所示。

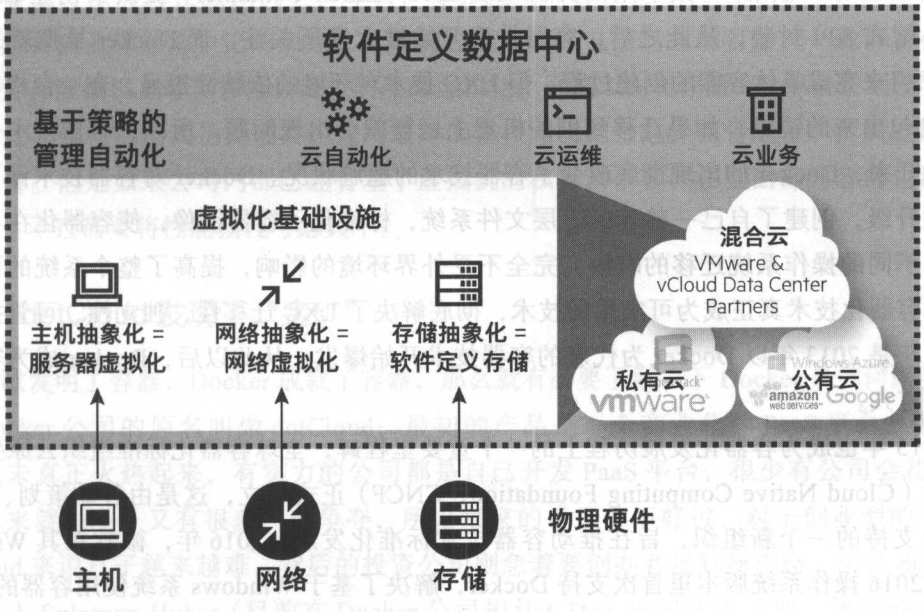


图1-6 虚拟化终极目标：软件定义数据中心

当数据中心中所有的功能都能通过软件去管理和控制时，未来世界就是一个软件的世界，我们通过软件定义和控制一切，就如同电影中所展示的情景：有了云计算和“软件定义一切”的基础之后，未来可能会出现真正控制整个世界的Skynet。

1.3 脱颖而出的容器技术

1.3.1 容器技术的历史

X86 上的虚拟机技术与容器技术基本上是并行独立发展的，初期虚拟机技术占上风，到了 2005 年，容器技术开始慢慢脱颖而出。容器技术的发展离不开谷歌的推动，我们一直以为 Docker 公司是容器技术的领头羊，但实际上谷歌才是容器技术真正的幕后推手。

谷歌的整个生产系统中一直没有使用虚拟机技术，而是全部采用容器技术来支撑。在 2015 年的 EuroSys 会议上，谷歌公布了多年以来的容器集群方面的秘密：谷歌早些年构建了一个管理系统，它可以用来管理集群、容器、网络以及命名系统。第一个版本被称为 Borg，后续版本称为 Omega。目前每秒会启动大约 7 000 个容器，每周可能会启动超过 20 亿个容器。利用多年在大规模容器技术上的实践经验和技术积累，谷歌构建了一个基于 Docker 容器的开源项目 Kubernetes，借此奠定了自己在容器界的霸主地位。

2006 年 KVM 开始发展，随后谷歌也开源了容器的底层核心技术 Cgroups，Cgroups 随后被纳入 Linux 内核中，接下来的开源项目 LXC（Linux Container）提供了创建 Linux 容器的一站式 API 封装，从此之后，容器技术开始被大家所关注，而 Docker 早期就采用了 LXC 项目来完成具体容器的创建过程。但 LXC 技术对环境的依赖性很强，在一台机器上用 LXC 打包出来的镜像，如果迁移到别的机器上运行就会出现问题。所以，容器技术一直没有流行开来，Docker 的出现彻底改变了容器技术的尴尬状况。Docker 对容器技术做了一次革命性升级，创建了自己一整套的分层文件系统，标准化了容器镜像，使容器化在不同的环境、不同的操作系统迁移的时候，完全不受外界环境的影响，提高了整个系统的可迁移性，使容器化技术真正成为可实用的技术，彻底解决了 LXC 迁移性、独立性、可管控性的问题。于是 2013 年以 Docker 为代表的容器技术开始爆发，从此以后，Docker 成为容器技术的代言人。

2015 年也成为容器化发展历程上的一个重要里程碑，全球容器化标准组织云原生计算基金会（Cloud Native Computing Foundation，CNCF）正式成立，这是由谷歌策划、Linux 基金会支持的一个新组织，旨在推动容器技术标准化发展。2016 年，微软在其 Windows Server 2016 操作系统版本里首次支持 Docker，解决了基于 Windows 系统使用容器的难题。这样一来，未来 Docker 不仅可以运行在 Linux 上，也可以在 Windows 上运行，我们可以看到，容器正在改变整个世界，不管是什么样的操作系统，容器技术都已经开始得到支撑。

提到容器，我们经常会碰到一个问题：容器和虚拟机真正的区别是什么？图 1-7 很好地解释了虚拟机和容器的区别。

图 1-7 左侧是虚拟机，我们看到，虚拟机需要一个完整的操作系统，而容器则是共享同一个宿主机的操作系统，因此容器是非常轻量级的，容器里面仅仅包括相关的用户代码

和所依赖的类库，因此容器也被称为是进程级的虚拟化。在一个操作系统上建立一个容器，实际上就相当于在一个操作系统上建立一个应用，因此它的启动速度和响应速度与虚拟机完全不在一个层面上。

另外，两者打包的镜像尺寸大小也不同，因为要包含一个完整的操作系统和各种类库，所以虚拟机的镜像非常大，一个生产环境中的虚拟机的镜像经常超过 1 GB，但是容器化镜像通常只有几十到几百兆。此外，镜像大小会大大影响整个系统弹性伸缩和快速部署的速度，例如，容器可以实现秒级的快速弹性伸缩，最主要的原因就是它的镜像尺寸比虚拟机要小很多，很快就能通过网络下载到目标机器并启动起来，而虚拟机可能仅仅下载镜像就需要几分钟，甚至更长的时间。

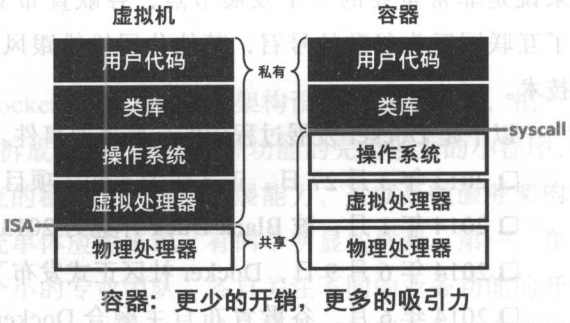


图 1-7 虚拟机和容器的区别

容器也可以部署在虚拟机上，那么，容器直接部署在物理机上还是部署在虚拟机上好？这是目前业界经常争论的焦点，目前的容器尤其是电信行业的容器，一般部署在裸机上，因为在虚拟机上部署容器会经过两层的虚拟化，网络 I/O 和存储 I/O 都会受到很大的影响，因此建议容器尤其是关键类的应用最好部署在裸机上，这样在管理上、扩展上就完全没有障碍。但是如果公司内部已经有基于虚拟机的私有云平台，在上面部署容器也没有太大问题，不过需要将性能损耗考虑其中。

1.3.2 dotCloud 发现了“金矿”

谷歌发明了容器，Docker 成就了容器，那么就有必要了解一下 Docker 是如何诞生的？

Docker 公司的原名叫做 dotCloud，最初的产品是一个商业化的 PaaS 平台，但 PaaS 市场从未真正火热起来，有实力的公司都是自己开发 PaaS 平台，很少有公司会花钱买。市场本来就很小，又有很多厂商争夺，所以每家的日子都不好过，对于创业型的小公司 dotCloud 来说日子越来越难，背后的投资公司则急着要创办者找到新出路，于是 dotCloud 的创办人 Solomon Hykes（目前在 Docker 公司担任 CTO）决定放手一搏，效法开源运动的精神，把公司在开发 PaaS 平台时为了方便采用 Linux Container 而研发的一套工具（即 Docker 的原始版本）开源出来。开源之后的这套新颖的容器工具受到了很多软件工程师的追捧，开发人员开发完程序之后只要用 Docker 打包成镜像交给测试人员，测试人员在本机就能使用这个镜像启动容器并进行快速测试，不同的版本被固化为不同的镜像，所以很容易进行回归测试，几个不同的镜像版本可以同时测试。由此开始，Docker 在业界赢得了很

好的口碑并迅速风靡开来。后来 dotCloud 公司就改名为 Docker，2014 年 6 月对于 Docker 来说是非常重要的一个发展节点，谷歌宣布支持 Docker，并且投资了 Docker 公司，有了互联网巨头谷歌的号召，其他公司纷纷跟风，越来越多的企业开始尝试并使用 Docker 技术。

以下是 Docker 发展过程中的一些重要事件。

- 2013 年 3 月 27 日，正式作为 public 项目发布。
- 2014 年 1 月，被 Black Duck 评选为 2013 年十大开源新项目。
- 2014 年 6 月 9 日，Docker 社区正式发布了 Docker 1.0。
- 2014 年 6 月，谷歌宣布自主融合 Docker 技术的云计算服务 Google App Engine 和 Google Compute Engine。
- 2014 年 9 月，获得 4 000 万美元的融资，此时已经累计融资 6 600 万美元。
- 2015 年 4 月，获得了 9 500 万美元的融资，已经确立了在第三代 PaaS 市场的主导地位。
- 2015 年，容器技术异军突起，其融合 DevOps 的敏捷特性备受业界关注，IBM、微软等传统厂商纷纷向 Docker 伸出橄榄枝。
- 2015 年 1 月，腾讯云计算公司对外宣布成为中国首家支持 Docker Machine 的云计算厂商，并将自身定位于 Docker 基础设施的服务商。
- 2015 年 6 月，Linux 基金会与行业巨头联手建立云原生计算基金会（CNCF）。

2015 年 6 月容器化标准组织 OCP 成立后，更多的大企业和创业公司开始拥抱 Docker，2015 年谷歌开源的 Kubernetes 奠定了其在容器领域微服务架构之王的地位，随后 Docker 公司的 Swarm 项目开始“模仿”Kubernetes，Mesos 则第一时间拥抱了 Kubernetes 这个重量级新事物。2016 年中国移动率先成功地在电信领域尝试大规模地部署和应用 Docker & Kubernetes 平台。Docker 项目的社区代码贡献者也由 2016 年年初的 900 多增加到目前的 1 200 多。

图 1-8 为我们展示了 Docker 生态圈的样貌，我们可以看到，操作系统厂商、虚拟化厂商、公有云厂商、服务器厂商和传统的 IT 巨头都成为 Docker 生态圈的一份子，而 IBM、微软、惠普等公司都是 Docker 主要的战略伙伴和投资人，容器技术不仅仅对运维的帮助很大，它对软件开发的生命周期管理也带来了很大的冲

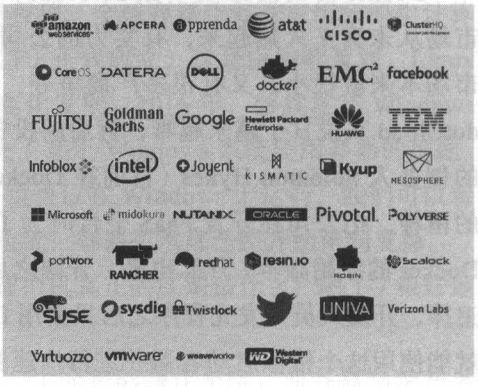


图 1-8 Docker 生态圈

击，大大促进了 DevOps 理念的落地。

1.3.3 容器技术带来的变革

容器技术带来了诸多的变革。首先，Docker 推动了微服务架构设计理念的落地，把一个原来很庞大的复杂的单体（单进程）应用拆成一个个基于业务功能的完全独立的小程序，并且分布式部署在一个集群中，以增加系统的稳定性、水平扩展能力，这就是微服务架构的核心思想和做法。微服务架构相对于传统单体应用来说，有两个明显的优势：第一，在开发上一个很大的团队完全可以拆成一个个小的专业团队，各自关注不同的业务功能的开发，因此系统的开发迭代、更新和升级就会变得非常敏捷；第二，由于微服务架构本身就是分布式架构，所以很容易实现系统的高可用以及快速弹性扩容能力，当某一个业务随着访问量的增大而出现性能瓶颈时，我们就能快速地对其进行弹性伸缩，增加服务实例数量，以改善整个系统的性能。

实际上微服务的理念很早就已经提出，微服务要求我们把一个完整的应用拆成一个个独立部署的微服务进程，并且部署在多个机器组成的一个集群中，每个机器上会部署很多微服务进程，不仅大大增加了系统发布、测试和部署的工作量，后续系统升级和运维管理的难度和复杂度也会大大提升。因此在缺乏自动化工具和相关平台支撑的情况下，微服务架构很难落地，长期以来只在一些大的互联网公司推行。

但 Docker 的出现改变了这一切。在 Docker 的帮助下，我们可以把每个微服务进程打包成独立的镜像，存储在统一的镜像仓库中，升级后的版本打包新镜像，采用新的 Tag 标签来区别于旧版本。只要写一个简单的脚本，以容器方式启动各个微服务程序，就能很快地在集群中完成整个系统的部署。还可以借助 Docker 引擎提供的 API，以编程方式来实现图形化的管理系统，一键发布系统、一键升级、自动修复系统等高级功能也都容易实现了。实际上谷歌开源的 Kubernetes 平台首次将微服务架构的思想贯穿到底，在 Kubernetes 的世界里，任何一个应用都由一个个独立的服务（Service）组成，一个具体业务流程实际上是由一个个服务串联在一起而完成的，部署应用的时候也按照服务的方式部署，不用关注服务到底会分布到哪些机器上，因为 Kubernetes 会自动调度 Service 对应的容器实例到可用的节点上，并提供高可用和弹性伸缩功能。实际上 Kubernetes 目前实现的功能特性早已超过微服务架构本身的要求，因此越来越多的公司开始使用 Kubernetes 平台打造自己的微服务架构系统。

其次，Docker 大大提升了软件开发和系统运维的效率，促进了 DevOps 体系的成熟与发展。

Docker 最大的特点是对应用的发布版做了一个标准化的封装，解决了应用的环境依赖

难题，并且不再需要安装部署过程。开发人员打包应用镜像之后就可以将镜像原封不动地转给测试人员，只要执行一个简单的启动命令，测试人员就可以在任意支持 Docker 的机器上成功运行应用程序并进入测试阶段。如果测试通过了就可以把这个镜像上传到镜像库中，随后运维人员可以直接从镜像库里把镜像拿出来并部署在生产集群中。这个过程完全可以建立一套标准化流程，因为每个环节传递的都是经过认证的标准化镜像，因此可以在后台通过一系列工具来控制整个流程的实现和度量。

通过一个流水线串联并驱动整个应用的开发生命周期过程，包括源码编译、镜像打包、自动部署或升级（测试环境）、自动化测试，以及运维阶段的监控告警、自动扩容等环节，这就是 DevOps 的实践思路。由于在这个过程中引入了 Docker 技术，从而很大程度上提升了系统运维的可管控性、可度量性、可监控性等重要指标，这就是 Docker 带来的第二个重要变革：进一步促进了 DevOps 的落地和发展。因此，在容器化平台改造建设完成之后，下一个重点目标就是建设 DevOps 平台，以促进整个软件的开发运维流程进一步向自动化、可管控性的目标迈进。

容器技术虽然是由 Docker 公司开源出来并发扬光大的，但背后是以谷歌为首的 IT 巨头在推进并且其已经成为标准规范，类似当年的 J2EE 组织，所以容器技术的影响力和影响范围会进一步扩张。

容器技术也是搭建企业 PaaS 平台以及新一代私有云最核心的技术，当前流行的 Kubernetes 和 Mesos，其底层都是以容器技术为基础搭建的，而且越来越多的企业正基于 Docker 和 Kubernetes 来改造或新建自己新一代的 PaaS 平台。

容器技术正在加速侵入 IT 的各个领域，并且影响和改变着整个生态圈，软件的设计理念、软件生命周期流程都因为容器技术的引入而发生革命性的变革。因此，可以毫不夸张地说，容器技术正在改变整个世界。

1.4 重新流行的 PaaS

1.4.1 PaaS 平台发展史

PaaS 平台的诞生虽然早于容器和虚拟机技术，多年来一直处于不愠不火的状态，但 PaaS 平台却因 Docker 的兴起而再次成为大家关注的焦点，新一代的 PaaS 平台开始以 Docker 为底层核心，更加接近 PaaS 最初的愿景。

最初的一批 PaaS 平台基本上以支持某种具体的编程语言为主，特别是很多公司自己开发的内部 PaaS 平台，这些 PaaS 平台通常提供了一套框架和工具以简化上层的应用开发。但实际上大部分平台都没有实现这个目标，反而增加了很大的限制，如开发语言的限制、框架特

殊的 API 接口的限制、支持的中间件种类的限制等。很多情况下，在 PaaS 平台上部署一个应用反而更加复杂，这些固有的缺陷导致 PaaS 平台不受欢迎，特别是不受开发人员的喜爱。

2007 年以后，出现了新一代的有一些影响力的 PaaS 产品，典型的代表有 Cloud Foundry 和 Heroku，它们开始基于最早的容器技术实现，采用了 LXC 技术，但由于彼时容器技术还没有普及，加之 LXC 的固有缺陷，因此这两款 PaaS 产品都卖得不好，最后 Heroku 被 Salesforce 并购，而 Cloud Foundry 则成为开源产品。前面提到，Docker 公司一开始也是做 PaaS 平台的，但与前辈们不同，Docker 公司在研发自己的 PaaS 平台的时候，突破性地解决了容器技术长期以来的缺陷，发明了 Docker 这个新一代的容器引擎并用于自家的 PaaS 产品，因此 Docker 在很多人看来实际上属于第三代 PaaS 技术的范畴。

自从 Docker 开源并流行开来以后，为 PaaS 平台注入了新的活力和发展方向，很多公司开始基于 Docker 开发自家的 PaaS 平台。最典型的 Kubernetes 也可以理解为一个思想超前的开源 PaaS 平台，红帽的 PaaS 平台——OpenShift——在 3.0 的时候也抛弃了原先的底层架构，而是彻底拥抱 Kubernetes。此外，Mesos 也在 0.2 版本之后加入了对 Docker 的支持。目前 Kubernetes 与 Mesos 这两个平台在大规模生产当中已经得到了应用和验证，这些案例分布在互联网企业、传统的金融电信行业，以及其他行业当中，这也说明这个容器技术已经非常成熟了，而且新一代的 PaaS 平台正在慢慢地提供商业化特点来满足不同行业的需求。

图 1-9 是 PaaS 的发展进程。

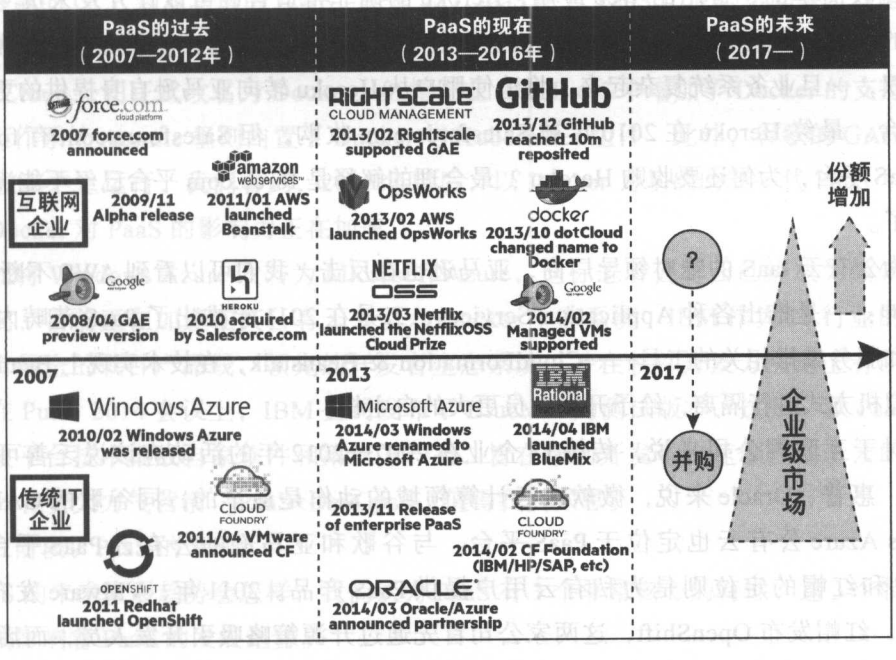


图 1-9 PAAS 的发展进程

PaaS 的发展阶段可以分为 3 个阶段。

第一阶段：PaaS 的过去（2007—2012 年）。

这个阶段最成功的其实是 Salesforce，Salesforce.com 作为最成功的 SaaS 公司，同时也是 PaaS 的鼻祖，因为 SaaS 应用在底层需要一个好的 PaaS 平台支撑，一方面他们需要 PaaS 平台来运行自家的 SaaS 软件，另一方面 PaaS 平台可用于支持用户的定制软件。2007 年，Salesforce 推出的 force.com 可以看作 PaaS 平台的“始祖”，它用来支持客户开发和部署定制软件，用户可以通过 Apex（与 Java 类似）和 Visualforce（UI）来开发运行在 force.com 上的应用，并与 Salesforce.com 应用进行集成。由于 force.com 平台采用了 meta data 驱动的架构来实现多租户机制，因此也有人把 force.com 称作“Metadata-PaaS”。

2008 年，谷歌为了与亚马逊的 AWS 争夺公有云市场，推出了 GAE（Google App Engine）平台，GAE 其实与 Salesforce 的 force.com 平台一样，也是一个 PaaS 平台，只不过谷歌的 GAE 建立在公有云技术之上，并且更加通用。2009 年 11 月，新浪也模仿了谷歌的 GAE 创建了自己的 PaaS 平台——SAE，一时间，国内其他互联网巨头纷纷效仿并推出自己的 ×AE 平台，但国内这些 PaaS 平台因为都只是立足于自己的开放平台战略，并不是真正的中立性开放，而且实力也有限，所以都没有发展起来。与此同时，国外的 Heroku 也成功模仿 GAE 并推出了运行于亚马逊 AWS 之上的公有 PaaS 服务，虽然 Heroku 依托 AWS 不断发展，并且深受 Ruby/Rails 开发人员的欢迎，但是 Heroku 并没有达到人们的预期。虽然对于比较简单的、常用的 Web 应用，Heroku 的确非常适合，可以让开发人员专注于业务本身的开发。但是 Heroku 在一定程度上限制了开发人员的选择，使开发人员失去了全栈的控制权，一旦业务系统复杂起来，将迫使用户从 Heroku 转向亚马逊自身提供的更底层的 IaaS 平台。最终 Heroku 在 2010 年被 Salesforce.com 收购，但 Salesforce.com 有 force.com 这个 PaaS 平台，为何还要收购 Heroku？最合理的解释是 force.com 平台已经不能满足其发展需求了。

作为公有云 IaaS 的绝对领导厂商，亚马逊也在反击，我们可以看到 AWS 不断向 PaaS 方向延伸：一是推出各种 Application Services，二是在 2011 年推出了 PaaS 范畴内的与资源管理和服务编排相关的工具——CloudFormation & Beanstalk，在技术实现上 Beanstalk 采用了虚拟机方式进行隔离，给予开发人员更大的自主权。

相对于互联网公司来说，传统 IT 企业在 2007—2012 年的活动应该说乏善可陈，较之 IBM、惠普、Oracle 来说，微软在云计算领域的动作是最早的，同谷歌的 GAE 一样，Windows Azure 公有云也定位于 PaaS 平台。与谷歌和亚马逊的公有云 PaaS 平台不同，VMware 和红帽的定位则是为私有云用户提供 PaaS 产品。2011 年，VMware 发布 Cloud Foundry，红帽发布 OpenShift，这两家公司首先通过开源策略吸引开发人员，而后尝试推出商业版本或者提供商业支持。这个策略很奏效，Cloud Foundry 在国内某种程度上成了

PaaS 的代名词，而 OpenShift 也有不少商业客户。

第二阶段：PaaS 的现在（2013—2016 年）。

谷歌在发布 GAE 后不久，就宣布 GAE 支持 Managed VMs 功能。与 Azure 的 VM Role 一样，这个功能给予 PaaS 开发人员完全的控制权。所以，未来 PaaS 与底层的 IaaS 之间将不会再有明显的界限与鸿沟，PaaS 可能成为 IaaS 的一个功能，它们会作为一个整体解决方案提供给上层的应用开发商。

作为亚马逊公有云上最成功的应用厂商，Netflix 在构建 Cloud-Native 应用上具有丰富的经验。在 Netflix 看来，AWS 提供的功能和它要开发的 Cloud-Native 应用需求之间存在一定的差距，这之间需要一个符合自己业务场景需求的 PaaS 平台来弥补，而最难的工作就是构建这个 PaaS 平台。从 2012 年开始，Netflix 自己开发的 PaaS 平台的一些组件逐步开源，目前在 PaaS 领域也有一定的影响力。

亚马逊方面则继续沿着 CloudFormation & Beanstalk 的 PaaS 之路深入发展，于 2013 年推出了 OpsWorks 服务。OpsWorks 将应用程序管理、可扩展性和性能结合在一起，并且进一步支持各种 DevOps 原则，如持续集成，用户不但可以控制如何部署代码，还可以使用 Chef 工具来实现自动化配置。OpsWorks 的推出引起了 RightScale 等亚马逊合作伙伴的不满，随后 RightScale 宣布支持谷歌的 GCE，不得不以支持 Multi-Cloud 的“新特性”来维持竞争力。

从谷歌的 GCE 到亚马逊的 AWS，公有云上 IaaS 融合 PaaS 已成必然趋势，这也是谷歌大力开源 Kubernetes 背后的真正动力所在。而 Docker 自 2013 年以来就非常火热，2013 年年底 dotCloud 公司正式改名为 Docker，红帽则在 RHEL 6.5 中增加了 Docker 的支持，在其下一代操作系统 Atomic 里则内置了 Docker 与 Kubernetes 组件。此外，谷歌的 GAE 也支持 Docker 在其之上运行，我们看到，2015 年以后，以 Docker 为基础的新一代 PaaS 平台正在兴起，Docker 对 PaaS 的影响力正在加速。

微软将 Windows Azure 改名为 Microsoft Azure，这标志着公有云已经成为微软的优先战略方向，目前正在加速在其公有云上引入 Docker 生态系统。IBM 作为 IT 行业的巨头之一，拥有最全的软件产品线、领先的开发者生态系统，并在软件开发方法论上有巨大的影响力。在 Pulse 2014 会议上，IBM 公司发布了 BlueMix 的测试版本，这是一款 PaaS 产品，通过开放平台技术集成自家软件和第三方产品，旨在帮助开发者快速创建基于云的企业应用，IBM 花费大力气营销“PaaS+DevOps”，值得我们深思。

第三阶段：PaaS 的未来。

PaaS 的未来发展趋势会怎样？我们认为 PaaS 有 3 个非常核心的特性，这 3 个特性一直并将持续影响着 PaaS 的发展趋势。

第一，多样性。用户的需求是多样的，同时由于历史原因，异构环境一直都是 IT 行

业必须面对的难题，这使得中间件领域的多样性比其他领域更为明显，所以不可能有一个PaaS服务可以满足所有人的需求，SaaS与PaaS融合、PaaS与IaaS融合是不可避免的。对于大型公有云提供商，提供一体化的基础服务是趋势，而对于创业公司，根据特定的细分市场提供差异化的PaaS & DevOps服务和工具也是机会。

第二，加速应用开发。PaaS的最终目标是为应用服务，加速应用开发是PaaS最重要的目标之一，但脱离App谈PaaS毫无意义。从App的热点方向看，未来的趋势有两个：一是遗留应用的集成和迁移；二是应用的移动化、社交化的加速发展。所以PaaS产品需要研究这些领域内的App所需要的特色服务，以便更好地切合市场。

第三，自动化。使用云服务和持续交付可以极大地加速业务创新，当PaaS提供的一切资源都以编程接口方式提供时，这意味着Full-Stack Automation成为可能。而当前兴起的DevOps理念的核心之一也是自动化，用户采纳云服务和DevOps有很强的正相关性，它们的结合是“银弹”，可以使软件开发和交付的效率得到前所未有的提高。此外，从DevOps角度看，如果说PaaS的终极目标是NoOps，那么任何有助于提高应用交付和管理效率的工具、服务都应该纳入PaaS的大范畴。

1.4.2 老牌的Cloud Foundry

Cloud Foundry是PaaS平台中最古老的产品，其特点是整个功能分层比较好，PaaS平台所应该具备的功能特点基本上Cloud Foundry都包含。自从Cloud Foundry基金会成立以后，IBM、惠普、SAP等巨头纷纷加入，有点抱团取暖的感觉，但对于IBM、惠普、SAP来说，作为后来者选择加入基金会却是一个无奈的选择，Cloud Foundry只是巨头们PaaS战略的一部分而已，所以这些巨头在Docker时代又各自有了新的方向，Cloud Foundry很可能逐渐没落。

目前不能在裸机上运行是Cloud Foundry的一个缺陷，因为当时的设计理念是遵循比较标准的IaaS、PaaS、Saas的分层构架而来的，底层必须有一个PaaS平台支撑，所以其必须基于开放的云构架才可以搭建PaaS。如图1-10所示的架构图，Cloud Foundry中位于最下面的BOSH层承担了PaaS平台和底层IaaS平台对接的功能，可以对接公有云或者私有云IaaS平台。

每个在Cloud Foundry中运行的应用都被称为Droplet，它也是容器化的。但Cloud Foundry没有使用Docker，因为Docker出来得晚，也没有用LXC，背后最主要的原因是LXC过于庞大，而Cloud Foundry只需要其中的一小部分功能，并且要求很容易对应用进行测试，所以Cloud Foundry自己开发了一套容器框架——Warden，其与Docker类似。负责Droplet管理的Droplet Execution Agent（DEA）调用Warden接口完成Droplet对应的

容器的创建和管理流程。DEA 部署在所有物理节点上（来自 IaaS 层的资源），这就形成了 DEA Pool——容器运行时的可调度资源池。

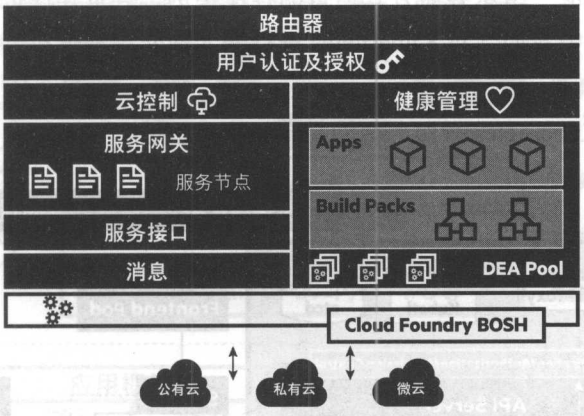


图 1-10 容器化 PaaS 平台：Cloud Foundry

Cloud Foundry 也提供了类似 Docker 的镜像打包工具，同时提供了包含 Java、PHP 等常见语言所开发的应用的标准化镜像打包工具。但 Cloud Foundry 没有提供更多的基础中间件的镜像，也没有提供类似 Docker 公共镜像仓库的机制以供大家分享和下载镜像，每个用户需要自己制作打包项目中所用到的各种镜像。众所周知，打包制作镜像这种任务的工作量和技术难度都很高，因此也导致了 Cloud Foundry 当前的状态——问世很久却一直没有在各个行业当中得到大规模推广并赢得用户口碑。

那么运行在 DEA Pool 里的应用如何与外部的服务互联互通？这就需要通过 Service Gateway 和 Service Connector 提供的功能来连接对内和对外服务。例如，一个外部的 Redis 服务要先注册到 Cloud Foundry 上，才能随后被 DEA Pool 里的应用所访问，因此 Cloud Foundry 不是一个非常完整的 PaaS 平台，这是它最大的缺陷，但是 PaaS 平台必须具备的功能，如智能路由、用户认证及授权、健康检查、云控制等都相对完备地实现了。因此，Cloud Foundry 作为 PaaS 的鼻祖产品，相对还是比较完善的。此外，目前 Cloud Foundry 最大的缺陷是没有用 Docker，但其新的版本也开始对 Docker 进行支持，未来 Docker 也可以运行在 DEA Pool 里。

1.4.3 Kubernetes & Mesos 新秀

首先，我们来说说微服务架构之王——Kubernetes。

Kubernetes 是目前 PaaS 领域里影响最大、最活跃的开源项目。用过 Kubernetes 的人都会有一个很深的体会：我们日常应用的设计、部署、扩容等环节，在 Kubernetes 里都能得

到很好的支持。

Kubernetes 中最重要的一个概念是 Service，这实际上是微服务架构理念，Service 就是一个微服务，背后对应一组从容器方式运行的程序实例，具备弹性扩容和自动故障恢复的能力。在后面的章节中我们会对 Kubernetes 进行深入的介绍和分析，因此这里就不做过多的介绍。

图 1-11 体现了 Kubernetes 微服务架构平台的核心思想。

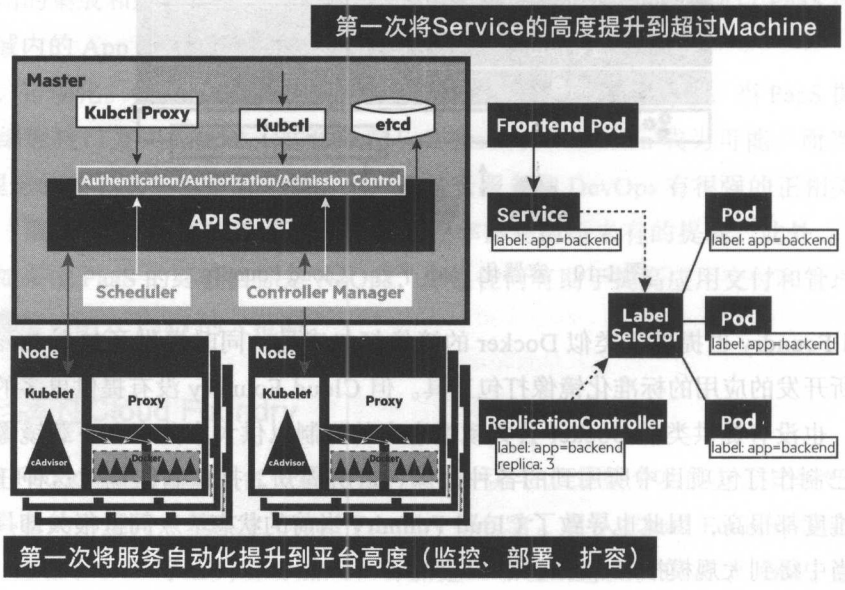


图 1-11 容器化 PaaS 平台——Kubernetes

Kubernetes 第一次将 Service 的高度提升到超过 Machine 的地位，我们不再关注应用部署的细节，只要规划好系统是由多少种不同的 Service 所组成的，每种 Service 需要部署多少个容器（Pod）实例才能满足性能要求，其他的就交给 Kubernetes 引擎了，它会根据当前集群资源的使用情况给出最佳的调度方案，并且自动完成容器实例的创建和部署过程，我们无须再花费脑力进行规划。

Kubernetes 的另外一大核心设计思想和优势是“高度自动化”。在 Kubernetes 的世界里，资源调度是自动化的，程序部署是自动化的，监控是自动化的，故障恢复是自动化的，服务平滑升级只要一键，服务扩容也只要一键，当前 Kubernetes 1.5 甚至已经初步实现了服务扩缩容的自动化。越深入学习和了解 Kubernetes，你就越发地惊叹谷歌惊为天人的巨大创造力。

其次，我们谈谈号称“数据中心操作系统”的 Mesos，如图 1-12 所示。

图 1-12　容器化 PaaS 平台——Mesos

从严格意义上说，Mesos 其实不属于一个完整的 PaaS 平台，而是一个先进的资源分配和任务调度系统。但考虑到资源管理也是 PaaS 的基本功能之一，并且很多通用应用也都已经能够在 Mesos 上安装并运行，所以我们也将其归到 PaaS 平台进行介绍。

Mesos 最初并不是针对容器来进行开发的，而是想对开源的大数据管理平台 Hadoop 进行资源的管控和调度，从而设计和研发的一个开源项目。最早出自加州大学伯克利分校。

Mesos 的设计理念分两层：下层是物理机器，组成资源池；上层是称为 Framework 的应用框架，每一个应用必须针对 Mesos 的 Framework 开发框架做定制化的开发后才能在 Mesos 上运行。为了支持 Docker 容器，Mesos 后来提供了 Marathon 这个 Framework。但 Marathon 并不是一个完整的微服务架构平台，缺乏负载均衡器、DNS 等必需的组件。

Mesos 的稳定性和健壮性在业界是比较认可的。例如，Twitter、MBNB 的数据中心都有由几千台机器组成的一个大的 Mesos 平台，现在也有很多的公司以“Mesos+Marathon”

为技术平台来搭建他们的容器化运行环境。Mesos 提出一个流行概念——DCOS，即把 Mesos 当成数据中心的一个操作系统，负责统一调度数据中心的每台物理机，其上则运行各种大数据系统，这样可以确保数据中心的物理资源得到更充分的利用。

Mesos 和 Kubernetes 这两个平台是可以完全融合在一起的，因为 Kubernetes 的重点是实现容器化应用的支撑和运行，而 Mesos 负责底层资源的分配和管控，相较于 Mesos 而言，Kubernetes 没有更细致的底层资源管控层，所以 Kubernetes 和 Mesos 之间的调度没有冲突，完全可以融合在一起实现整个资源调度的平台。

具体的做法是 Kubernetes 可以作为一个 Framework 运行在 Mesos 之上，替换原来的 Marathon 实现容器化应用的支撑，而底层的资源则完全交给 Mesos 来负责管控和统一调度。实际上，Mesos 中运行 Kubernetes 的方案就是采用了这个思路。

那么，Kubernetes 是否可以替代 Mesos 呢？

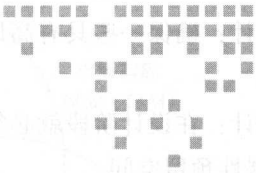
这个问题目前还没有清晰的答案，但我们看到，Kubernetes 除了继续完善微服务架构的高级功能之外，也开始进入任务调度的新领地。从本质上来说，任务调度系统的设计和开发难度要远远小于微服务架构系统，如果 Kubernetes 想往这方面深入发展，凭借谷歌的领导力和实力，有朝一日超越 Mesos 也不是不可能的事情。

最后，我们来简单谈谈 PaaS 与 Iaas 融合的问题。

OpenStack 原来是 Iaas 平台系统，但是随着容器技术的发展，OpenStack 也开始支持 Docker。最近由谷歌牵头发起的 Magnum 子项目的目标是在 OpenStack 上直接支持容器，当然其主要目的是支持自家的 Kubernetes 平台，这样一来，既提升了 OpenStack 的应用场景，又借着 OpenStack 这艘大船，进一步扩展了 Kubernetes 的地盘，是一箭双雕的好事。

未来基于开源体系搭建的私有云（行业云）的技术栈目前看来非常清晰了，底层的 IaaS 平台可以用 OpenStack 来实现主机、存储、网络的虚拟化和集中管控，上层则搭建基于容器化的新一代 PaaS 平台（Kubernetes、Mesos 等）来支撑上层应用，再辅以一些自动化运维工具和定制开发的 Web 管理系统，即可形成一套低成本同时又具备技术领先性的私有云。

在后面的章节中，我们将具体聊一聊微服务架构、DevOps、Docker、Kubernetes、Mesos，以及企业级容器云在电信行业的应用实践。



第 2 章 Chapter 2

微 服 务

我们所负责的系统是否适合进行微服务化改造？如果适合我们应该如何着手将现有系统一步步微服务化？在本章我们将围绕微服务的概念、特点、设计模式，以及如何将一个单体拆分为微服务这几个主要的关键点，来了解一下什么是微服务。

2.1 为何要做微服务

2.1.1 架构设计新理念：做好隔离

无论是微服务还是单体服务，都是一种项目的架构方式，项目架构首先要确认架构原则是什么，以及目标是什么，答案很简单，架构原则需要根据公司的目标来制定，也就是说架构原则应该与公司的发展愿景和使命相符。架构目标就是盈利，盈利的方法有很多，通过降低成本换取利益最大化是盈利的一个主要途径。而如何能实现降本增效的目的呢？其实无论是单体服务还是微服务，都可以从可用性、可扩展性、质量、成本、效率、响应时间这几个维度入手。微服务也是把单体解构之后，变成一个个构建小、发布快的小服务，从而提高可用性，增强可扩展性，最终完成降本增效的目的。

在这里，我们列出 15 个最常用的架构原则，这些原则并不是在所有的设计中都必须涉及，而是供客户剪裁，由客户根据项目的特性增加或者删除某项或某几项原则。我们概括地讨论这些原则，并着重考虑其与微服务架构设计的相关性。

1）*N*+1 设计：确保系统发生故障时，至少有一个冗余的实例。

2）回滚设计：确保系统可以回滚到以前发布过的任何版本。

3）禁用设计：确保一些具有高风险的系统功能能够通过开关来禁用，这能为修复赢得时间。

4）监控设计：在设计阶段就必须考虑监控，而不是在实施完成之后。监控做得好，将为系统的可扩展性预留空间。

5）设计多活的数据中心：确保系统可在地理上隔离灾难和危机。

6）使用成熟的技术：只用确实好用的技术。

7）异步设计：只有在确实有必要的情况下才使用同步设计，以增加系统的可扩展性。

8）无状态系统：只有在确实有必要的情况下才使用状态，状态耗费资金，降低处理能力、可用性和可扩展性。

9）水平扩展而非垂直升级：永远不要依赖更大、更快的系统。

10）设计至少要有两个步骤的前瞻性：在扩展性问题上提前考虑好下一步的行动计划。

11）非核心的购买：如果不是最擅长的，也提供不了差异化的具有竞争优势的功能，则直接购买。

12）使用大规模量产的商品化硬件：只有在确实有必要的情况下才定制硬件。

13）小构建、小发布、快试错：不断迭代，让系统不断地成长。

14）自动化：设计和构建自动化的过程，如果机器可以做，就不要依赖人。

15）隔离故障：实现故障隔离设计，通过断路保护避免故障传播和交叉影响。

以上就是我们常用的一些架构原则，我们也可以用图 2-1 这样的形式直观地了解微服务的架构原则。在技术高速发展的今天，架构设计理念也在发生着变化。例如，平均修复时间和平均故障间隔时间是做系统时必然会面对的两个指标，那么这两个指标哪个更重要一点？其实这个问题没有标准答案，历史经验更多是关注平均故障间隔时间，选用很健壮的数据库，选择高可用的中间键，购买小机，提高高可用性，增加系统的安全设置，目的都是尽量少出错、不出错，让错误在更早的时候及时得到处理，这一切都是为了提升平均故障间隔时间。但到了现阶段，当我们购买的 X86 越来越多，甚至系统所占的机房和数据中心不再是一个的时候，我们的理念也随之发生了变化：我们认为出错是必然的。有运维经验的人都知道，如果有上百台的 X86，那么出错是必然的，很难像原来的小机时代，能够实现尽量不出错。

既然出错是必然的，那么我们就需要在设计初期考虑如何能将出错造成的影响降到最小，这时做好隔离就尤为重要了。无论是从虚拟机、容器层面，还是到现在从数据和应用两个层面切分做微服务，所有的这一切都是为了实现隔离，让一个点出错产生的影响最小，并更快恢复，最大限度地降低出错对业务的影响，这也是目前我们在架构设计理念的显著变化。

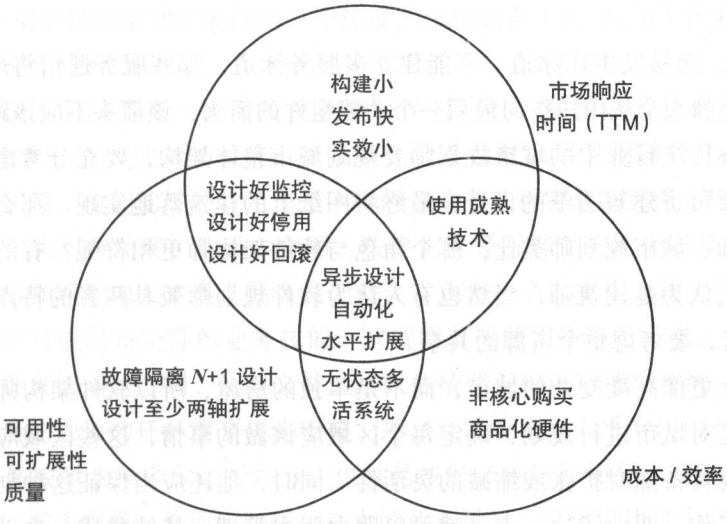


图 2-1 微服务架构原则

微服务是实现隔离的重要手段，但故障隔离不是免费的，也未必是便宜的。那么何种情况下需要优先考虑通过微服务实现隔离呢？答案取决于该系统的特定需求、增长率、不可用率和客户期望等一系列因素。如果微服务故障隔离做好了，后面会得到很好的回馈。

在这里，我们提供一些简单的原则来帮助大家制定故障隔离方案。

1）泳道与盈利原则：一定要确保与盈利关系最密切的事情和可能失败及有需求限制的其他系统适当地隔离，并创建泳道。例如，如果是一个商务系统，影响业务最关键的就是订单和收费这两个与钱关系最紧密的模块，我们需要考虑将其从硬件、中间应用层和最上层的网络层都实现隔离。

2）隔离频繁出错组件：如果有的模块特别容易出错，那么这个模块也具备率先考虑通过微服务实现隔离的特质。

3）泳道与天然隔离：客户边界是最好的泳道隔离锁，这种分割沿着客户的边界实施。分割通常先在存储或数据库层完成，再创建一条从请求到数据库并从数据库返回客户的完整泳道。

隔离会耗费成本和资源，共享将获得成本效益，所以我们需要权衡隔离。在大多数情况下，单一泳道可运行多个应用。但如果某个服务特别忙，那么可以给它分配一个专用泳道；如果大部分服务利用率都很低，那么可以把它们都分配到一个泳道。

如何进行故障隔离，有几条重要的原则。

1）原则 1：绝不共享或者尽量减少共享。包括共享网络、共享硬件、共享 URL、共享数据库等一切资源，泳道共享越少，其故障隔离度就越高。

2）原则 2：不跨越泳道。同步通信从不跨越泳道，如果跨越了泳道，那么边界划分就

会不正确。

3）原则 3：交易发生在泳道。不能建立多服务泳道，那些服务通信将违反原则 2。如果我们在白板上画一个从用户指向最后一个功能组件的箭头，该箭头不应该跨越泳道。

我们都了解传统行业中的城市规划师要规划城市整体架构，要充分考虑水、电各种区块；而建筑师要负责建筑图纸的设计，最终将图纸上的建筑落地实现。那么，如果将软件架构师与建筑师、城市规划师类比，哪个角色与软件架构师更相符呢？有的人认为是城市规划师，也有人认为是建筑师，当然也有人认为软件规划师兼具两者的特点，因为做系统要考虑很多因素，要考虑整个资源的共享。

现在的系统更像高度复杂的城市，而不是单独的房屋，所以软件架构师更像城市规划师，他的职责是对城市进行规划，确定每个区域应该做的事情，这些区域应当达到统一的规范要求，又要具备随时扩张或缩减的灵活性。同时，他还应当保证这种划分适合专业人员在对应区域工作。明白了这一点，就可以明白版本管理、持续集成、自动化测试、自动化发布、服务治理、详尽监控等“磨刀工夫”的价值，没有这些工作，就谈不上微服务的质量和保障级别，也就无法驾驭微服务的体系。

由此，也很容易明白软件架构师在这个时代所要面对的挑战。一方面，软件架构师没有现成的足够强大的集成工具，只能靠一堆“稀松平常”的工具组装出整体可靠的系统；另一方面，要深入理解业务，把业务拆散成边界清晰的概念，以“高内聚低耦合”的服务分而治之，还必须考虑实现的限制——“高内聚低耦合”的原则人人都知道，如果没有可靠的分布式事务管理机制，就不得不把并非“高内聚”的模块聚合起来，但又要担心业务边界模糊的风险。RESTful 固然优雅，但有时候又不得不使用 RPC 通信，所以又要提防 RPC 带来的强绑定，客户端、服务器端同步更新等很多问题。

这一切设计、权衡、决策并没有成型的理论和学说可以依靠，通常只能完全依赖软件架构师的经验、理解、思考。所以困难很大，风险也很大，如果做得好，收益也是非凡的。而这恰恰是软件架构师的价值所在。

2.1.2 如何利用扩展立方体切分应用和数据

谈到如何切分应用，免不了要谈扩展立方体。拓展立方体并不是特定应用于软件的，而是全行业很多领域都可以适配。

图 2-2 是一个三维轴线立方体，我们将 3 个轴线的交点称为初始点，其坐标值 x=0、

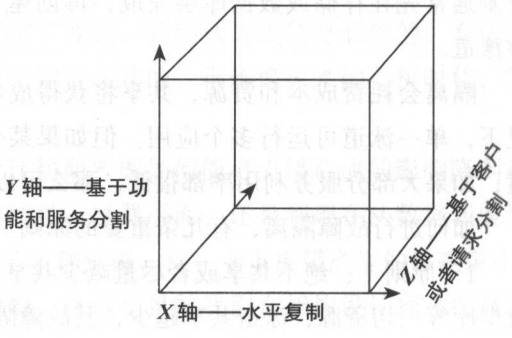


图 2-2　扩展立方体

y=0 和 z=0，每个轴线描述扩展性的一个维度。坐标原点（0，0，0）代表任何系统、流程、产品中最小的可扩展性。

举例来说，如果要在一座大厦里创建一个打字社，随着业务量的扩大，如何进行扩展？如图 2-2 所示，可能最初只有一个店员，业务量增大后又招聘了几个店员，这就是按照 *X* 轴进行扩展，他们的工种、工作是一样的，这是一种完全的克隆。*Y* 轴是什么？当业务继续增大时，我们会发现，信函、合同、法律文件等所需要的模板以及上下文背景是不同的，可能还需要熟悉相关文案，那么我们就会发现由一个人承担上述全部工作是不现实的，因为一个人不可能具备全部的业务技能，这就需要按照工作进行分类，把人分成几组，不同的人分别负责信函、合同、法律文件，这就是按照 *Y* 轴进行切分扩展。随着业务的发展，我们会发现诸如领导的任务需要优先安排，那么又需要添加人手专门服务领导或 VIP 客户，这实际上就是按照业务来源进行切分，也就是按照 *Z* 轴进行切分。所以无论做什么事，涉及如何扩展、增大规模，如何应对持续加大的业务量时，都可以用这个立方体进行适配。

回到微服务领域，如何将扩展立方体应用于切分应用就比较好理解了，如图 2-3 所示。

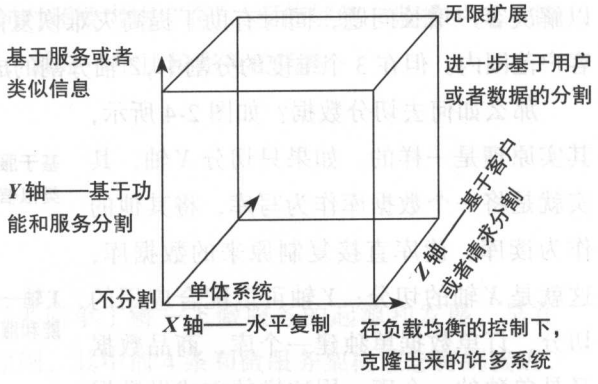


图 2-3 为扩展分割应用

X 轴表示把相同的数据或镜像分散在多个实体上，通常依赖于复制，并对节点数设限制。*X* 轴的分割往往是最廉价的分割实施，但它对客户或数据增长的扩展没有帮助。

Y 轴表示基于服务、资源或数据关系的意义分离和分散数据。*Y* 轴分割有助于帮助解决数据扩展问题，并且有助于故障隔离，但对交易增长的扩展没有 *X* 轴分割有效，并且比 *X* 轴分割的实施成本要高。

Z 轴表示基于请求的属性分割数据。例如，基于客户或产品 ID 分割数据，一般通过查找获取分割数据。*Z* 轴分割有助于交易增长和数据增长的可扩展性，并且能实现比 *Y* 轴分割更均衡的负载均衡，同时也有助于故障隔离，但同样成本高昂。

也就是说，单体系统是原点，我们想从原点慢慢扩展，最初可能是水平复制，克隆出许多相同的系统，前面加一个负载均衡，这就是 *X* 轴的切分。微服务一般都是基于功能的，按照商品、订单、物流、支付分解成不同的模块，这就是 *Y* 轴，按照我们的动作去切分。*Z* 轴是按照来源切分，例如，系统来源的请求是多个省份的，就可以按照省份切分，或者用一个算法统一将来源进行切分。

这里需要补充的是我们在做 *Y* 轴和 *Z* 轴切分的同时都可以配 *X* 轴，再加一个负载均衡，

后面就可以克隆出很多的订单、商品模块，或者按照某一个数据库水平切分之后，某些量大的会辅之以更多应用、进程进行支撑，所以都会有 X 轴的切分。并且我们一般的做法是先切分 X 轴，同时切分 Y 轴再配 X 轴，再往前一步配 Z 轴。最终的目的是 X、Y、Z 轴都做了切分之后，进入一个无限扩展的愿景。

X 轴的分割代表把相同的工作或应用镜像分配给多个实体，X 轴的分割很好地解决了交易量的扩展问题，并且实施成本较低，但它无法解决越来越多的数据所产生的问题，也无法解决软件实施、平台或产品的复杂性问题。Y 轴分割代表把工作职责基于“动词”或动作进行分割并分配到多个实体。Y 轴分割是为了解决复杂的代码和数据集不断增长所带来的问题。其目的是为分割后的交易处理建立故障隔离，减少交易响应时间。但由于团队需要重写服务或至少分解原有的紧密耦合的单体应用，所以拆分成本比较高。同时，拆分带来了更多的运维工作，特别是在配置管理和系统管理方面的复杂度也进一步增加。Z 轴分割代表按照客户、位置分配工作，可以建立故障隔离，而且可以沿着客户的边界扩展。Z 轴分割可以解决客户增长问题，同时有助于提高灾难恢复能力，把事故的影响限制在一部分特定的客户范围内。但在 3 个维度的分割中，Z 轴分割的成本往往最大，并且增加了运维的复杂性。

那么如何去切分数据？如图 2-4 所示，其实原理是一样的。如果只切分 X 轴，其实就是将一个数据库作为写库，将其他的作为读库。读库直接复制原来的数据库，这就是 X 轴的切分；Y 轴可以配合应用的切分，订单数据单独建一个库，商品数据又是单独的一个库，用这样的方式做数据的切分；Z 轴更好理解，直接将库表按照分片或者其他方式进行区分，同时按照扩展立方体切分数据和应用。

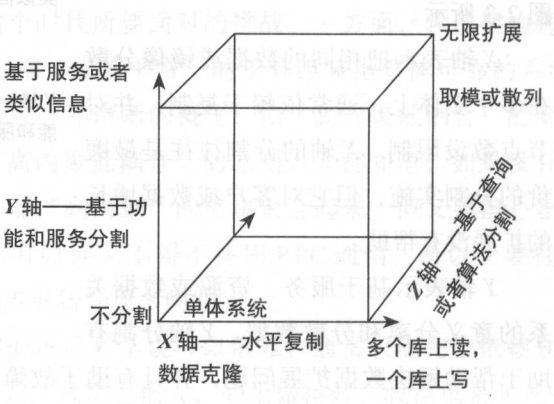


图 2-4　为扩展分割数据

2.1.3　时间考虑和融会贯通

在了解了如何利用扩展立方体切分 X 轴、Y 轴、Z 轴，以及如何切分数据和应用之后，我们常常面临着同时要切分应用和数据的情况，那么是先切分应用，还是先切分数据呢？决定什么时候选择哪种方法或哪个轴进行扩展，是一门科学，也是一门艺术，决定以“应用为主”的架构或是以“数据为主”的架构进行分割的时候也是如此。

如果业务清晰，如何切分应用就很清楚了，可以把这些比较独立的或者无状态的、可扩展的模块切割出来，同时数据也跟着切分了。例如，按照订单、支付、物流单个模块，

后面的数据也可以直接一并进行切分。从操作的层面看，一般建议先切分数据，这是因为数据在操作面比应用的切分更简单，可以直接按照某种业务场景进行分库，之后的工作会比给一个单体重新写代码或加很多补丁所做的操作要少，这样一来，可以把数据先分库，这样比较好回退。

那么具体应该何时采用 X 轴分割、Y 轴分割或者 Z 轴分割呢？理论上并没有具体的时间表可供参考，但在实践中有一些经验可循。在理想情况下，一个技术或者架构团队会根据实际情况选择适合他们的扩展方法，就高交易量、低数据需求、高读写比率的系统而言，最具成本效益的分割方法就是 X 轴分割。但在客户数据增长、功能复杂度增加和交易量增长同时发生的情况下，该系统可能需要进行 3 个轴的分割。

综上所述，在当前环境下如何做个好架构，简单来说，就是“拆与合”，先“拆”后“合”。当我们把一个个单体拆解成独立的、小的微服务时，这些微服务可以独立上线、独立发布，有独立的周期，有自己的团队，但如果“合”没做好，还是会遇到很多问题。所以从总体的架构层面来看，我们在“拆”的同时还要考虑“合”，要能够统一监控、统一日志、统一出错管理，能“拆好”又能“合好”，就是一个比较完美的架构。

2.2 微服务概要介绍

2.2.1 微服务架构原理

在了解了为什么要做微服务之后，我们来简单了解一下微服务的起源和发展。早在 1994 年 Mike Gancarz 就提出了 9 条著名的原则，其中前 4 条和微服务架构理念特别接近，微服务就像把 UNIX 哲学应用到了分布式系统。

- small is beautiful（小即是美）：小的服务代码少、bug 少、易测试、易维护，也更容易不断迭代完善，精致进而美妙。
- make each program do one thing well（一个程序只做好一件事）：一个服务只需要做好一件事，专注才能做好。
- build a prototype as soon as possible（尽可能早地创建原型）：尽可能早地提供服务 API，建立服务契约，达成服务间沟通的一致性约定，至于实现和完善可以慢慢地做。
- choose portability over efficiency（可移植性比效率更重要）：服务间的轻量级交互协议在效率和可移植性两者间，首要依然考虑兼容性和移植性。

2011 年 5 月在威尼斯召开的软件架构研讨会（QCon）上，“微服务”这一术语被讨论用来描述参与者一直在探索的一种常见的架构风格。2012 年 5 月，在 QCon 旧金山会议上，该研讨会决定使用“微服务”作为最合适的名字。可见微服务其实不是凭空产生的，它自

有其历史渊源。

国际著名的 OO 专家、敏捷开发方法的创始人之一马丁·福勒先生对微服务架构做出了自己的定义，他认为微服务架构是一种将一个单一应用程序开发为一组小型服务的方法，每个服务运行在自己的进程中，服务间通信采用轻量级通信机制（通常用 HTTP 资源 API）。这些服务围绕业务能力构建并且可通过全自动部署机制独立部署。这些服务共用一个最小型的集中式管理，服务可用不同的语言开发，使用不同的数据存储技术。

其实微服务架构是一种思想、一种方法论，不会完全依赖于哪些技术、哪些平台，马丁·福勒提出将单一的应用程序开发，转变成一组小型服务的方法，这其实就是“拆”的过程，然后再阐述“合”的过程。

那么微服务有哪些特征？

首先，每个微服务都在自己的进程中；其次，服务和服务之间除了轻量级的通信机制，也就是 API 的调用之外，不能有其他的调用方式，不能直接访问别人服务的数据，不能直接有一个类似库表之间的调用，只能是轻量级的通信机制；最后，这些服务围绕业务能力构建，并且全部可以独立部署，可以自动化实现一个个小的构件，然后这些服务共用一个最小型的集中式管理，服务可以用适合的语言和数据库。这就是一个微服务。

如图 2-5 所示，微服务的诞生绝非偶然，是多重因素推动下的必然产物，它的出现具备天时、地利、人和，而互联网就是驱动微服务出现的源动力。

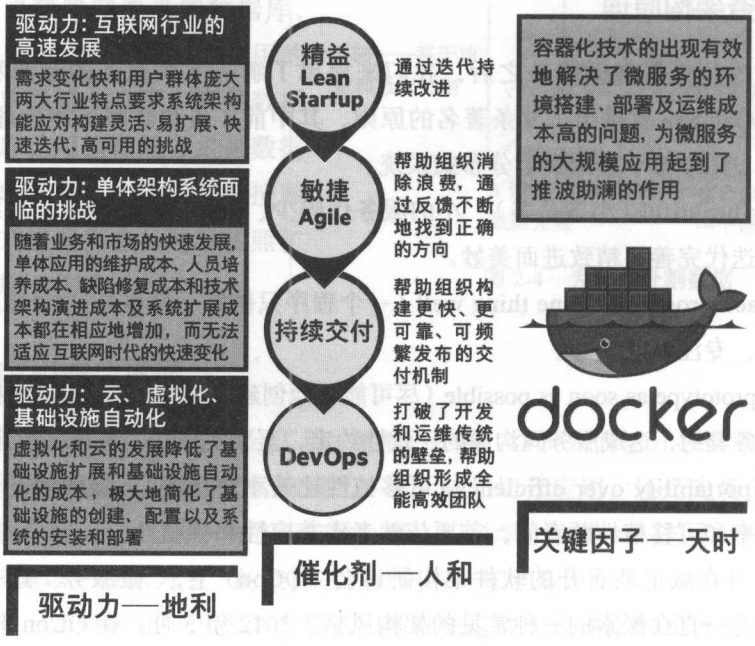
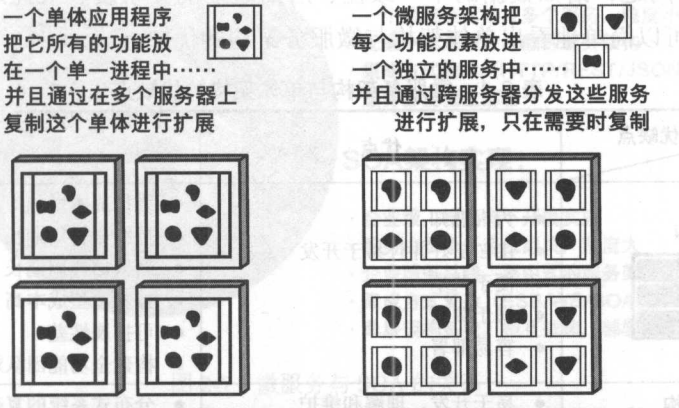


图 2-5　微服务的产生

无论是微服务还是容器化，都是为了解决伴随互联网蓬勃发展所出现的问题，只有拆分成一个个小的服务，才能更好地管理，让每个团队更灵动，从而促进团队之间的竞争。同时，所有的竞争都要靠人来完成，而人的管理如果没有先进的手段保证，将无法进行。纵观国际也是一样，亚马逊的云公司之所以做得这么好，是由于一种天然因素的驱动。亚马逊要做全球的生意，如果不谈管理难以实现，亚马逊才会想办法做各种架构、自动化，同时又将其成果开源。而国内情况也类似，阿里巴巴的云技术在国内领先，也是因为其业务的量变引发了质变，催化了阿里的云技术发展。

目前的状况是自媒体公司一半以上的资金量要买机房做压缩、传输，做 IDC，IDC 占互联网公司成本的一半，另外一半资金用来建设团队、做运营、做市场推广。基本上任何一家互联网公司的模式都是这样的。因为量变引发了质变，才会有今天的敏捷、持续交付、微服务，Docker 提供了很好的封装、打包，让所有技术有了更好的支点，所以微服务、容器化、DevOps 如同一个铁三角，密不可分。

简单来说，微服务就是一些协同工作的小而自治的服务，如图 2-6 所示，微服务与单体架构有不同的风格。



特征	描述
自治	每个微服务是一个功能单元，围绕业务模块和通用功能为每个功能提供 API 接口
隔离	每个微服务是一个部署单元，能够作为单元被修改、测试和部署，而不会对其他组件造成影响
弹性	微服务无状态，能够根据需要进行水平扩展和收缩
适应性	微服务能够适应失败、容错和高可用，应为面向失败的设计考虑
响应	微服务能对一定时间内的请求作出响应
面向消息	面向同步和异步消息，建立各组件之间的界限，这是确保松耦合、隔离，并提供处理错误消息的方法
去中心化	服务的调用和通信是直接的、无中心的
可组合	应用由多个微服务构成
自动化	微服务的生命周期使用自动化管理，包括部署、构建、测试、预发布、生产和分发

图 2-6 微服务与单体架构的风格

（1）小而专注

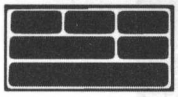
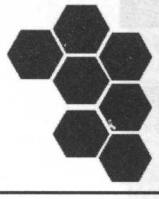
在单体系统中，随着新功能的不断增加，代码库会越变越大，以至于修改和系统集成的难度随之上升。而为了避免这些问题，微服务高内聚低耦合的特性应运而生。把因相同原因而变化的东西聚合在一起，把因不同原因而变化的东西分开来。微服务将内聚性应用在独立的服务上，根据业务的边界来确定服务的边界，可避免代码库过大衍生的问题。但独立性带来的好处越多，管理就越复杂。

（2）自治

一个微服务是一个独立的实体，它可以独立部署、独立修改，服务之间通过暴露 API 来进行通信，API 的实现技术应避免与具体技术相关，以保证技术的选择不受限制。自治性其实就是要求服务与其他的服务之间很好地解耦。判断是否解耦有一个黄金法则，即你是否能够修改一个服务并对其进行部署，而不影响其他任何服务，如果是，那么就可以实现很好的解耦。

具体来讲，每个微服务都包括自治、隔离、弹性、适应性、响应、去中心化、可组合、自动化等特性，单体的微服务能够实现更好的部署、更好的测试，但“合”的时间可能会更长，会有很多问题，诸如数据的不一致性、后面通信的复杂度、彼此之间的依赖等。通过表 2-7，我们可以简单地看出单体架构与微服务架构的优缺点。

表 2-1 微服务架构与单体架构的优缺点

架构 \ 优缺点	优点	缺点
单体架构 	• 为人熟知 • IDE 友好并且易于开发 • 便于共享 • 易于测试 • 容易部署	• 维护成本增加 • 持续交付周期长 • 新人培养周期长 • 技术造型成本高 • 可扩展性差 • 构建全功能团队难
微服务架构 	• 易于开发、理解和维护 • 比单体应用启动快 • 局部修改很容易部署，有利于持续集成和持续交付 • 故障隔离，一个服务出现问题不会影响整个应用 • 不会受限于任何技术栈	• 分布式系统的复杂度：性能、可靠性、异步、一致性、工具 • 运维成本增长 • 必须具备部署自动化功能 • DevOps 运作组织复杂 • 服务间依赖导致管理复杂 • 服务间依赖导致测试复杂

那么，微服务和 SOA 是什么关系？

SOA 是一种设计方法，它通过提供服务向外提供一系列功能，服务之间通过网络调用，而非采用进程内调用的方式进行通信。SOA 可以应用于庞大的单块应用程序，从而提高软件的复用性，如多个终端共享一个服务。大多数 SOA 的学说更倾向于理论和概念的层面，

关于服务的“粒度”定在哪个层级，服务如何落地，如何保证可用性等问题始终没有取得广泛的共识，对于软件所依赖的环境，SOA 也很少涉及，但软件的运行是离不开外部环境的。所以 SOA 的学说虽然热门，但真正做到了、做好了的例子非常有限。

在这种局面下，微服务应运而生了。承接 SOA 的概念，微服务将系统按照业务责任划分为彼此独立的多个服务，既保证了概念的清晰和自洽，又保证了系统的灵活性、伸缩性。面对杂乱不可靠的现实，又从实现上注重每个服务的自治性，也就是能独立部署，具备自动化、可观察、故障隔离、自动恢复等特性，由此提供高可用保障。

如图 2-7 所示，微服务在原来 SOA 的基础上更进了一步，原来系统之间用 SOA 理念集成起来，但却没有定义每个系统具体什么样。微服务的特性决定了要对系统重新进行定义，系统要自治，要独立部署，要有隔离性，只能和别人用轻量级的通信方式，可以有自己专属的技术和数据。在系统内部要切成一个个微服务，每个微服务要具备的特性都要定义清楚，会比原来的 SOA 有更好的操作性。

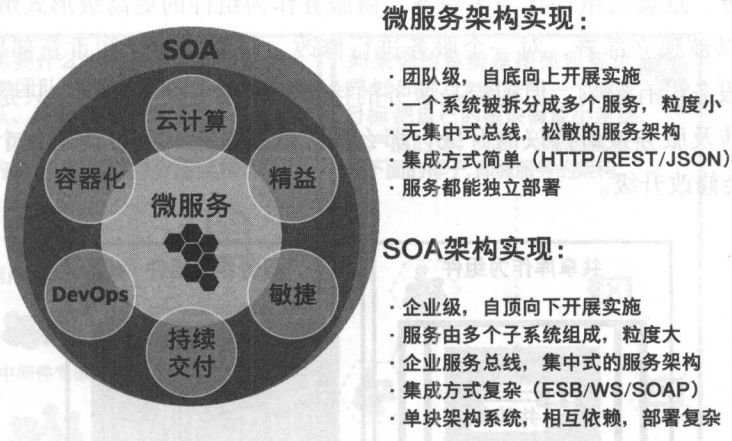


图 2-7 微服务与 SOA 的关系

在 SOA 尚且没有完整落地的时候，对它有继承、有更新、有颠覆的微服务极大地增加了开发人员的挑战。

首先，服务要拆得足够小，又不至于太小，这样才能保证伸缩性并隔离故障；其次，不能因为服务“小”就降低保障级别，维护一堆“小服务”的保障级别是极其麻烦的事情；再次，要做到上面这一切，无论是从理论还是从可依赖的软硬件系统上，都没有现成的低成本解决方案；最后，因为维护的是动态的“服务”，传统静态的“代码所有权”和“机器所有权”的划分不再有效，它们已经融合为统一的“服务所有权”，它属于开发人员、运维人员以及所有相关人员的共同体，这又会给团队配合与分工带来挑战。

2.2.2 微服务的特性

微服务有九大特性，我们逐一来了解一下。

（1）特性一：“组件化”与“多服务”

如图 2-8 所示，传统实现组件的方式是通过库（library），库和应用一起运行在进程中，库的局部变化意味着整个应用的重新部署。通过服务来实现组件，意味着将应用拆散为一系列的服务运行在不同的进程中，那么单一服务的局部变化只需重新部署对应的服务进程。

将软件库（libraries）定义为这样的组件，即它能被链接到一段程序，且能通过内存中的函数来进行调用。然而，服务（services）是进程外的组件，它们通过诸如 Web Service 请求或远程过程调用这样的机制来进行通信。原来编写程序，无论是使用 C 语言还是 Java，都会引入很多组件，一个应用程序外部引入很多成熟的库。到了微服务，实际上我们要把原来引入的这些组件也单独变成一个个的服务，可能原来是一个日志的组件，或是消息的组件，在这之上我们做了某个业务，到了微服务我们会把日志、消息、应用过程都作为一个个单独的服务，这就是组件化与多服务。微服务作为组件的更高级形式解决了一个关键问题：服务可以被独立部署，对一个服务进行修改，只需要构建和重新部署这一个服务，对系统的其他服务没有影响，但前提是服务的接口协议没有发生变化，只是对服务内部进行升级。如果涉及服务接口协议的升级，那么情况就要复杂一些，调用这个服务相关的其他服务可能也会修改升级。

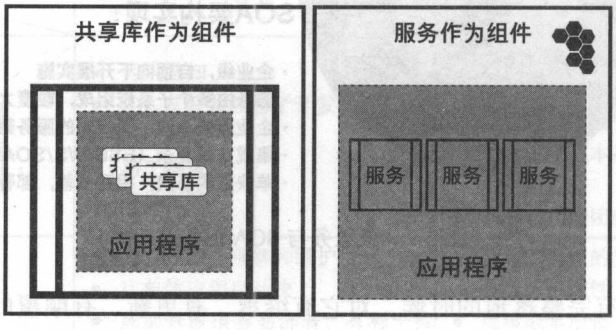


图 2-8 微服务特性一：“组件化”与“多服务”

（2）特性二：围绕“业务功能”组织团队

1967 年，《哈佛商业评论》拒绝了 Mel Conway 提交的一篇论文。一年之后，Conway 的论文最终被确定为“Conway’s Law”（康威定律）。Conway 在加利福尼亚理工学院获得物理学硕士学位，在凯斯西储大学获得数学博士学位。毕业之后，他参与了很多知名的软件项目，如 Pascal 编辑器。他在职业生涯中观察到一个现象：软件团队开发的产品是对公司组织架构的反映。用通俗的说法就是组织形式等同系统设计。这里的系统按原作者的意思

并不局限于软件系统。最初这篇文章显然不敢自称定律（law），只是描述了作者自己的发现和总结。后来，在 Brooks Law 著名的《人月神话》中，引用了这个论点，并将其“吹捧”成了现在我们熟知的“康威定律”。

康威定律中提到围绕业务功能组织团队，简单来说就是什么样的团队产生什么样的架构，这也是微服务所有架构的根基。如图 2-9 所示，具体来说，就是如果团队是按照业务功能组成的多个小微团队，那么每一个小组开发出来的必然是微服务；如果团队是按照前端、中间键、数据库这样的方式构建的，那么开发出来的必然是单体。

我们来回顾一下康威定律，其中提到了 4 个观点。

1）定律一：组织的沟通方式会通过系统的设计表达出来。简单来说，假设一个团队有 N 人，那么会有多少条沟通线路？答案是 $N \times N-1/2$。无论是做项目还是做产品，要解决的最大问题就是沟通问题。沟通的好坏在很大程度上决定项目的成败。所以有多少条沟通线路，可能最后就会有多少个模块。沟通的问题会带来系统设计的问题，进而影响整个系统的开发效率和最终产品的结果。

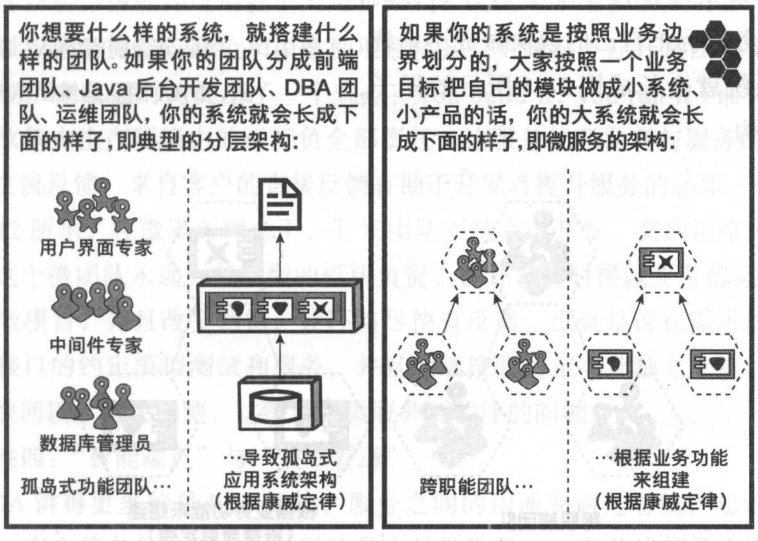


图 2-9 微服务特性二：围绕“业务功能”组织团队

2）定律二：即便有再多的时间，一件事也不可能做得完美，但总有时间做完一件事。我们可以这样理解，做项目时必须围绕项目目标在给定的时间和资源内，把一件事情做到最好。建议放弃打造完美系统的想法，而是通过不断的试飞，发现问题，确保问题发生时系统能自动复原，而不追求飞行系统的绝对正确和安全。这就是持续集成和敏捷开发在微服务中得以应用和推广的原因。此外，这和互联网公司维护的分布式系统的弹性设计是一个道理。对于一个分布式系统，几乎永远不可能找到并修复所有的 bug，单元测试覆盖

1000% 也没有用，错误流淌在分布式系统的血液里。解决方法不是消灭这些问题，而是容忍这些问题，在问题发生时能自动恢复。在微服务组成的系统中，每一个微服务都可能出错，这是常态，我们只要有足够的冗余和备份即可，即所谓的弹性（resilience）设计或者叫高可用（high availability）设计。

3）定律三：也就是前面我们讲到的围绕业务功能组织团队。线性系统和线性组织架构间有潜在的一致同态特性，它们是一一对应的。这是康威第一定律组织和设计间内在关系的一个具体应用。更直白地说，想要什么样的系统，就搭建什么样的团队。如果团队分成前端团队、Java 后台开发团队、DBA 团队、运维团队，系统就会“长”成图 2-10 所示的样子。

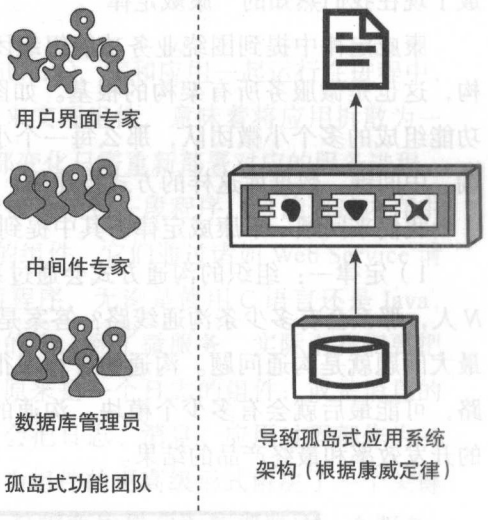


图 2-10 团队分为前端团队、Java 后台开发团队、DBA 团队运维团队时的系统

相反，如果系统是按照业务边界划分的，即按照一个业务目标把自己的模块做成小系统、小产品，大系统就会长成图 2-11 所示的样子，即微服务的架构。

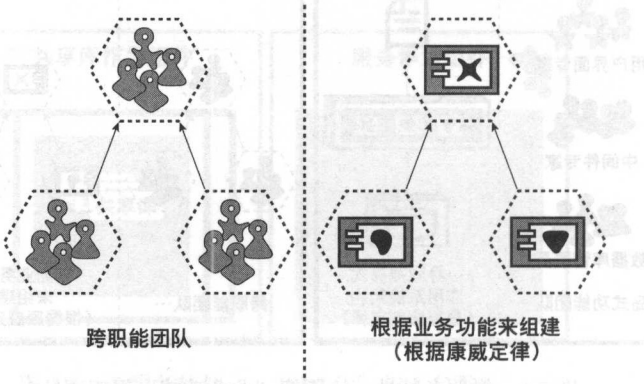


图 2-11 微服务的架构

微服务的理念是团队间应该服务内部高内聚，服务之间低耦合。定义好系统的边界和接口，在一个团队内全栈，让团队自治，如果团队按照这样的方式组建，将沟通的成本维持在系统内部，每个子系统就会更加内聚，彼此的依赖耦合能变弱，跨系统的沟通成本也就能降低。

4）定律四：大的系统总是比小的系统更倾向于分解。前面说了，人是复杂的社会动物，

人与人的沟通非常复杂。但是当我们面对复杂系统时，又往往只能通过增加人力来解决。这时，我们的组织一般是如何解决这个沟通问题的呢？答案是分而治之。业务需求驱动系统越长越大，当达到一定规模时，无论我们有没有微服务架构这个理论，我们首先想到的就是把大模块拆成小模块，把大的团队拆成小团队，只有分解了，才能把人的自由性发挥出来，才能够做好后面的事情。

人与人的沟通是非常复杂的，一个人的沟通精力是有限的，所以当问题太复杂需要很多人解决的时候，需要通过拆分组织来提升沟通效率。组织内人与人的沟通方式决定了他们参与的系统设计，管理者可以通过不同的拆分方式带来不同的团队间沟通方式，从而影响系统设计。如果子系统是内聚的，与外部的沟通边界是明确的，能降低沟通成本，对应的设计也会更合理高效。复杂的系统需要通过容错弹性的方式持续优化，不要指望一个大而全的设计或架构，好的架构和设计都是慢慢迭代出来的。

（3）特性三：关注产品而不是项目，也就是“谁构建谁运行”

传统的应用开发都是基于项目模式的，开发团队根据一堆功能列表开发出一个软件应用并交付给客户后，该软件应用就进入维护模式，由另一个维护团队负责，开发团队的职责结束。而微服务架构建议避免采用这种项目模式，更倾向于让开发团队负责整个产品的全部生命周期。亚马逊对此提出了一个观点：You build it，you run it（谁构建，谁运行）。开发团队对软件在生产环境中的运行负全部责任，让服务的开发者与服务的使用者（客户）形成每日的交流反馈，来自客户的直接反馈有助于开发者提升服务的品质。

如图 2-12 所示，在微服务理念下，开发团队不是完成开发、交到运维团队之后就不再管了，而是这个微团队永远为其所做的模块负责，在开发的时候就要考虑运营和运维需求，关注产品和微项目，并且改变回馈，让环路尽快有反馈。也就是说在需求阶段就要把测试做好，按照接口的约定模拟测试和服务，并且后续按照约定彼此独立上线、独立测试，尽快回路、尽快回馈、尽快反馈，这些都是微服务要关注的问题。

（4）特性四：“智能端点”与“傻瓜管道”

原来 SOA 讲得更多的是 ESB 架构，服务之间的沟通要通过总线，总线除了通信，还要负责路由、安全等各种控制。这种做法最大的弊端是：一般总线都是商用的产品，因此会有相对的语言依赖度、数据依赖度，造成对总线的依赖度很重。

如图 2-13 所示，对于微服务架构，我们不像原来会有一个很“重”的 ESB 总线，微服务架构抛弃了 ESB 过度复杂的业务规则编排、消息路由等。服务作为智能终端，将所有的业务智能逻辑在服务内部进行处理，而将服务间的通信尽可能地轻量化，不添加任何额外的业务规则。所以这里的智能终端是指服务本身，而哑管道是通信机制，可以是同步的 RPC，也可以是异步的 MQ，它们只作为消息通道，在传输过程中不会附加额外的业务智能。

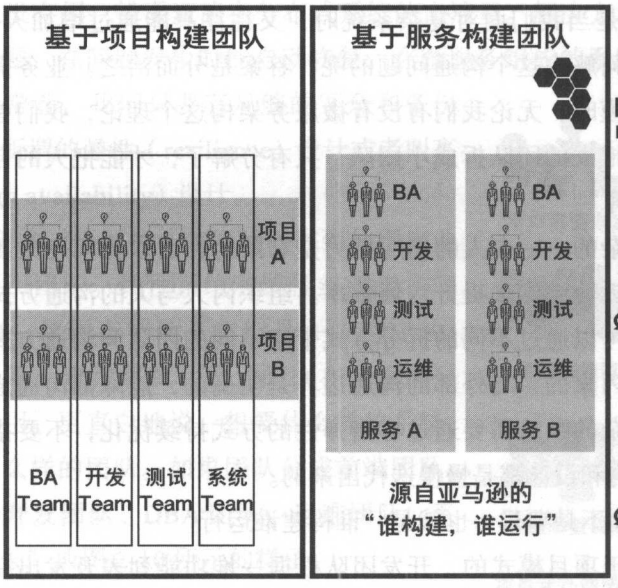


图 2-12 微服务特性三：关注产品而非项目

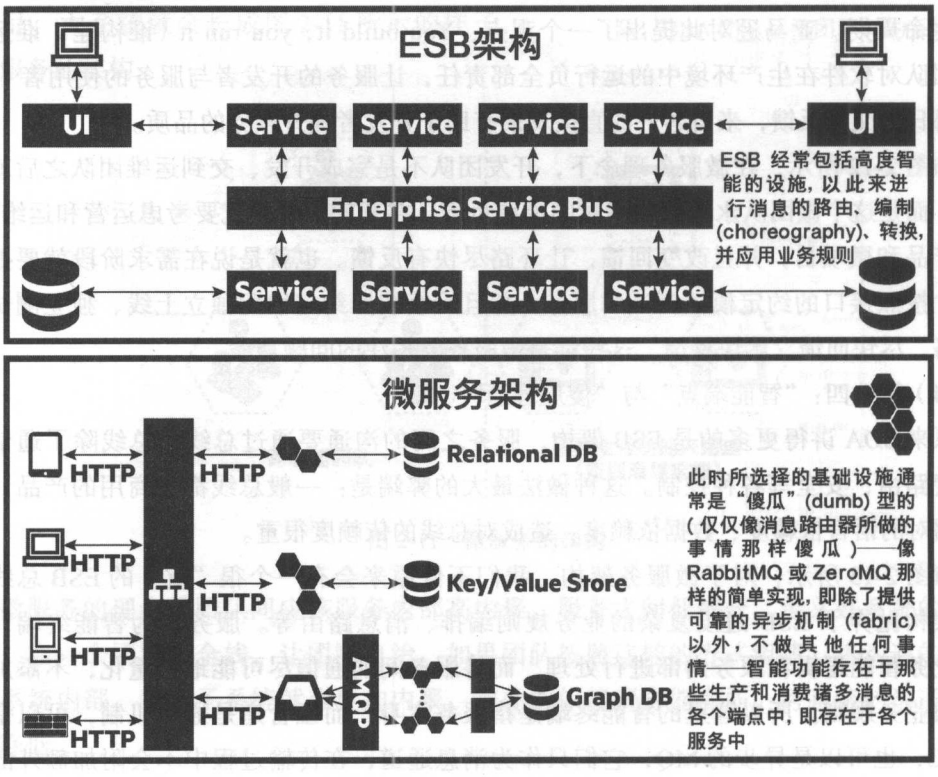


图 2-13 微服务特性四：“智能端点”与“傻瓜管道”

最常用的第二种协议是通过一个轻量级的消息总线来发送消息。此时所选择的基础设施通常是“傻瓜”型的（仅仅像消息路由器所做的事情那样傻瓜），如 RabbitMQ 或 ZeroMQ 那样的简单实现，即除了提供可靠的异步机制（fabric）以外，不做其他任何事情，智能功能存在于那些生产和消费诸多消息的各个端点中，即存在于各个服务中。

（5）特性五与特性六：去中心化地治理技术 & 去中心化地管理数据

去中心化包含两层意思：技术栈的去中心化与数据去中心化，如图 2-14 所示。

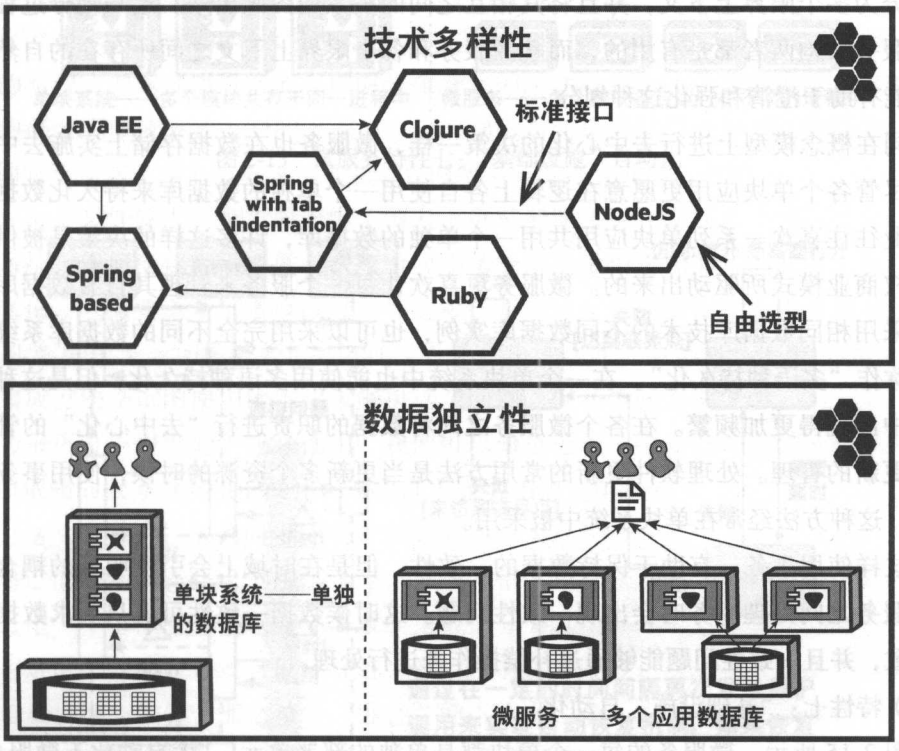


图 2-14 微服务的特性五、特性六

每个服务面对的业务场景不同，可以有针对性地选择合适的技术解决方案。但也需要避免过度多样化，结合团队实际情况来取舍，如果每个服务都用不同的语言技术栈来实现，则维护成本很高。

不像传统应用共享一个缓存和数据库，每个服务独享自身的数据存储设施（缓存、数据库等），这样有利于服务的独立性，隔离相关干扰。

去中心化地管理数据，其表现形式多种多样。从最抽象的层面看，这意味着各个系统对客观世界所构建的概念模型将彼此各不相同。当在一个大型的企业中进行系统集成时，这就是一个常见的问题。例如，对于“客户”这个概念，从销售人员的视角看，就与从支持人员

的视角看有所不同。从销售人员的视角所看到的一些被称之为“客户”的，或许在支持人员的视角中根本找不到。而那些在两个视角中都能看到的事物，或许各自具有不同的属性。更糟糕的是，那些在两个视角中具有相同属性的事物，或许在语义上有微妙的不同。

上述问题在不同的应用程序之间经常出现，同时也会出现在这些应用程序内部，特别是当一个应用程序被分成不同的组件时。思考这类问题的一个有用的方法就是使用领域驱动设计（Domain-Driven Design，DDD）中的“限界上下文”的概念。DDD 将一个复杂的领域划分为多个限界上下文，并且将其相互之间的关系用图画出来。这一划分过程对于单块和微服务架构两者都是有用的，而且在服务和各个限界上下文之间所存在的自然的联动关系，能有助于澄清和强化这种划分。

如同在概念模型上进行去中心化的决策一样，微服务也在数据存储上实施去中心化的决策。尽管各个单块应用更愿意在逻辑上各自使用一个单独的数据库来持久化数据，但是各家企业往往喜欢一系列单块应用共用一个单独的数据库，许多这样的决策是被供应商的各种版权商业模式所驱动出来的。微服务更喜欢让每一个服务来管理其自有数据库，其实现可以采用相同数据库技术的不同数据库实例，也可以采用完全不同的数据库系统，这种方法被称作“多语种持久化”。在一个单块系统中也能使用多语种持久化，但是这种方法在微服务中出现得更加频繁。在各个微服务之间将数据的职责进行“去中心化”的管理会影响软件更新的管理。处理软件更新的常用方法是当更新多个资源的时候，使用事务来保证一致性，这种方法经常在单块系统中被采用。

像这样使用事务，有助于保持数据的一致性。但是在时域上会引发明显的耦合，这样在多个服务之间处理事务时会出现一致性问题。这时候数据一致性可能只要求数据在最终达到一致，并且一致性问题能够通过补偿操作来进行处理。

（6）特性七：“基础设施”自动化

如图 2-15 所示，微服务的每一个模块都是单独的部署单元。“无自动化不微服务”，自动化包括测试和部署。单一进程的传统应用被拆分为一系列的多进程服务后，意味着开发、调试、测试、监控和部署的复杂度都会相应增大，必须要有合适的自动化基础设施来支持微服务架构模式，否则开发、运维成本将大大增加。

（7）特性八：“容错”设计

如图 2-16 所示，微服务架构采用粗粒度的进程间通信，引入了额外的复杂性和需要处理的新问题，如网络延迟、消息格式、负载均衡和容错，忽略其中任何一点都属于对“分布式计算的误解”。容错设计就是要做好日志、做好监控，能够最快地检测出故障，尽快地恢复故障。

具体的应对措施包括：

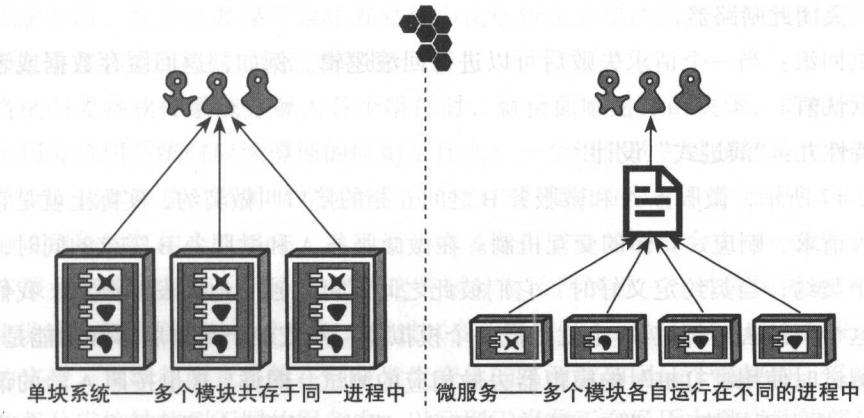


图 2-15 微服务特性七：“基础设施”自动化

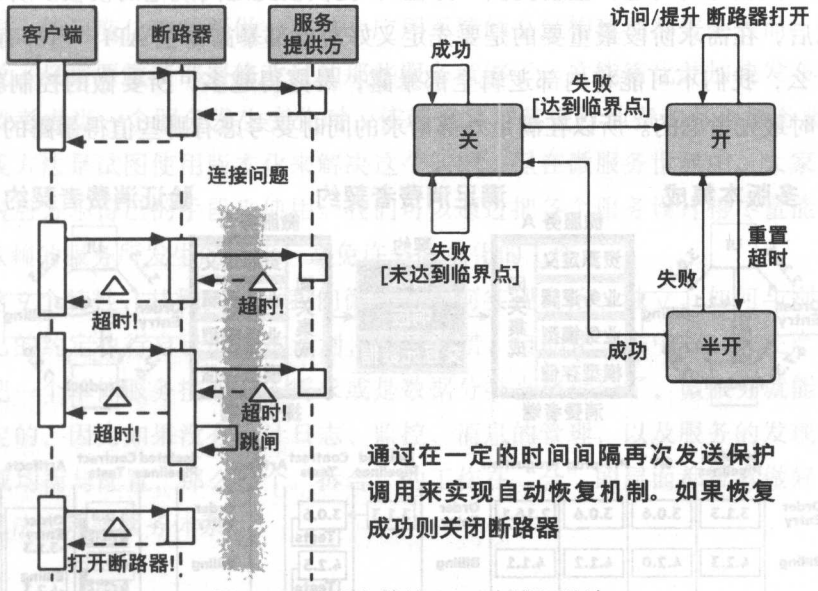


图 2-16 微服务特性八：“容错”设计

- 网络超时：在等待响应时，不要无限期地阻塞，而是采用超时策略，使用超时策略可以确保资源不会被无限期地占用。
- 限制请求的次数：可以为客户端对某特定服务的请求设置一个访问上限，如果请求已达上限，就要立刻终止请求服务。
- 断路器模式（circuit breaker pattern）：记录成功和失败请求的数量。如果失效率超过一个阈值，触发断路器使得后续的请求立刻失败。如果大量的请求失败，就可能是这个服务不可用，再发请求也无意义。在一个失效期后，客户端可以再试，如果成

功，关闭此断路器。

- 提供回滚：当一个请求失败后可以进行回滚逻辑。例如，返回缓存数据或者一个系统默认值。

（8）特性九：“演进式”设计

如图 2-17 所示，微服务 A 和微服务 B 之间互相的接口叫做契约，实际上就是它们之间接口的输入请求、响应、具体的交互机制。在做微服务 A 和微服务 B 需求的同时，一定要先定义这个契约，当契约定义好时，它们彼此之间就可以独立地开发和测试。我们作为 A 可以按照这个契约去“打桩”，也就是做一个模拟器，不依赖于具体版本和功能是否可用，最初上线测试时使用“打桩”的模拟器去做相应的测试。同样，B 也按照 A 给的请求反应去“打桩”，它也不依赖 A，即 B 可以独立开发、测试。只要契约没有变化，它们彼此之间就相互独立，彼此之间没有依赖，就可以按照各自的版本去做自己的更新；当契约发生变化时，一定要做相应的沟通，重新提供“打桩”，提供彼此之间规范的模板。所以当我们做了微服务以后，在需求阶段最重要的是要先定义好对外部暴露哪些 API，内部有什么功能，外部暴露什么，我们不可能将内部逻辑全部暴露，暴露得越多，所要做的控制就越多，这是做微服务时最先考虑的。所以在满足外部请求的同时要考虑有哪些值得暴露的 API。

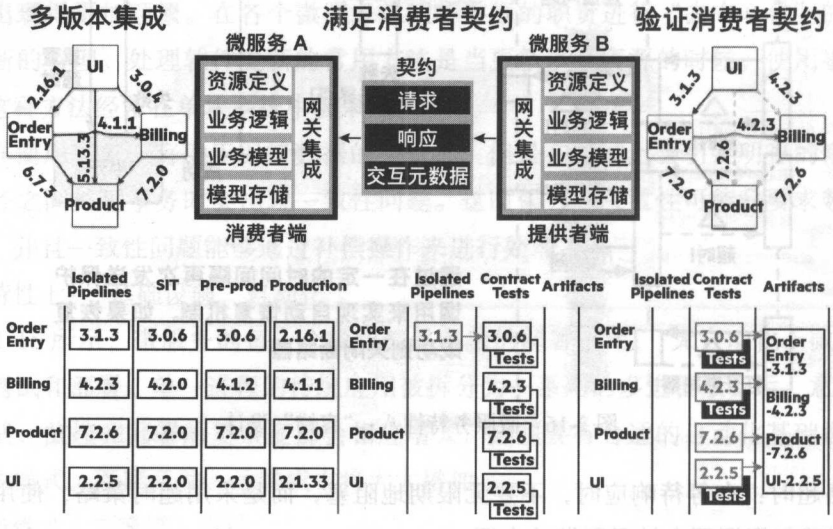


图 2-17 微服务特性九：“演进式”设计

一旦采用了微服务架构模式，那么在服务变更时要特别小心，服务提供者的变更可能引发服务消费者的兼容性被破坏，要时刻谨记保持服务契约（接口）的兼容性。一条普适的健壮性原则（伯斯塔尔法则）给出了很好的建议：Be conservative in what you send, be liberal in what you accept（发送时要保守，接收时要开放）。按照伯斯塔尔法则的思想来设

计和实现服务时，发送时要保守意味着要最小化地传送必要的信息，接收时更开放意味着要最大限度地容忍冗余数据，保证兼容性。

每当试图要将软件系统分解为各个组件时，就会面临这样的决策，即如何进行切分，我们决定切分应用系统时应该遵循的原则是什么？一个组件的关键属性是具有独立更换和升级的特点，这意味着，需要寻找这些点，即想象着能否在其中一个点上重写该组件，而无须影响该组件的其他合作组件。事实上，许多做微服务的团队会更进一步，他们明确地预期许多服务将来会报废，而不是守着这些服务做长期演进。这种强调可更换性的特点，是模块化设计一般性原则的一个特例，通过“变化模式”（pattern of change）来驱动并进行模块化的实现。大家都愿意将能在同时发生变化的东西放到同一个模块中。系统中很少发生变化的部分应该被放到不同的服务中，以区别于当前正在经历大量变动的部分。如果发现需要同时反复变更两个服务时，这就是它们两个需要被合并的一个信号。

把一个个组件放入一个个服务中，增大了更精细地实现软件发布计划的机会。对于一个单块系统，任何变化都需要做一次整个应用系统的全量构建和部署。然而，对于一个个微服务来说，只需要重新部署修改过的那些服务就够了，这能简化并加快发布过程。但缺点是必须要考虑当一个服务发生变化时，依赖它并对其进行消费的其他服务将无法工作。传统的集成方法是试图使用版本化来解决这个问题。但在微服务世界中，大家更喜欢将版本化作为最后万不得已的手段来使用。我们可以通过把各个服务设计得尽量能够容错，来应对其所依赖的服务所发生的变化，避免许多版本化的工作。

以上这 9 个特性，从侧面告诉我们微服务如何实现自治、独立，如何与别人沟通，如何按照自己的约定执行自己的版本计划，如何容错、自动化、去中心、关注产品、尽快响应。那么把一个单体服务按照业务需求或是数据分类直接拆分了，微服务就能上线了吗？答案是否定的，因为如果没有设计日志、监控、消息的管理，以及服务的发现机制和整个服务的负载均衡与配置，那么整个“拆合”的工作在“合”的层面都没有做好，在这种情况下我们无法实现微服务体系。

2.2.3 完整微服务系统包含的功能

一个完整的微服务系统应该包含哪些功能？其实微服务的关键不仅仅是微服务本身，而是系统要提供一套基础架构，这种架构使得微服务可以独立地部署、运行、升级，不仅如此，这个系统架构还让各个微服务之间在结构上实现“松耦合”，在功能上则表现为一个统一的整体。这种所谓的“统一的整体”表现出来的是统一风格的界面、统一的权限管理、统一的安全策略、统一的上线过程、统一的日志和审计方法、统一的调度方式、统一的访问入口等。

从图 2-18 我们可以看出，一个完整的微服务系统应该包括一个基座，要有日志和审计、监控和告警、消息（轻量级 MQ 或 HTTP）、注册发现、负载均衡、事件调度，只有把

这些基座都完成了，才可以在上面部署一个个的业务微服务。基座之上，我们要考虑外部和服务之间如何暴露 API，外部请求只能通过服务网关来访问微服务，所有关于微服务相应的流控、安全、访问健全管理都会在这个网关上完成。

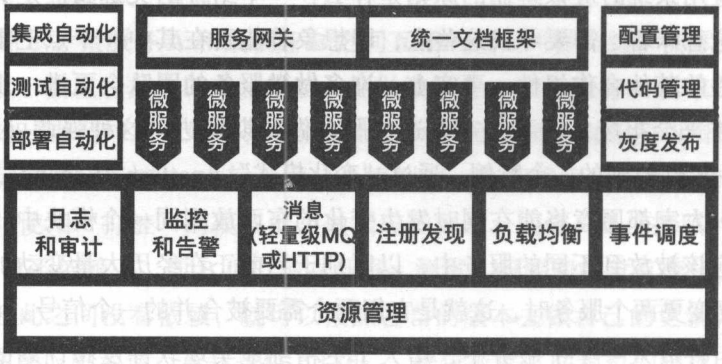


图 2-18 完整的微服务所包含的功能

我们对外暴露了很多 API，那么外部如何找到关于这些 API 的定义？ API 的每个动作都会有什么样的输入和输出？配合 API 网关，会有一个统一的文档框架，这个文档框架应该是开源的，也是一个工具集，大家可以不再去找拥有这些微服务的个人，而是直接找公司的主体，一般运营商都会对外有一个开放的门户，其中会定义对外开放了哪些能力，通过文档框架可找到开放了哪些能力。

一个基座、一个访问控制是整体微服务的根基。同时要有集成自动化、测试自动化、部署自动化，要有一个自动化流水线，这是为微服务团队的开发和集成部署服务的。另外，配合配置管理、代码管理、灰度发布，也要有相应的工具集。一般来讲，一个生产系统差不多会有 3 ～ 4 套环境，即一套开发环境、一套上线环境和一套对外真正的生产环境，有时还会有对外的性能测试环境。如果 3 ～ 4 套环境部署在不同的机房和不同的 IDC 里面，而每一次的配置如果是通过人工去完成的，那么为了一次性能测试，可以想象我们可能需要好几天时间来提前准备环境、测试和数据。但如果我们将配置预先写好，底层也是基于容器化的资源管理，那么为某一种测试建立一套环境就会简单很多，也无须用 3 ～ 4 天的时间预先准备一个环境，配置实际上主要是为了在这种多环境下实时部署。

无论是敏捷还是 DevOps，我们认为最大的好处是上线和停机，7 × 24 小时，一天做 100 个版本，但不能忽视其根本——灰度发布，我们必须先把环境进行分区，当想发布新版本的时候，会把请求路由到其中一个区，升级另外的分区，升级完成后再把请求升级到新分区，所以至少要有一个基本的灰度发布分区。总体来说，我们最终希望实现的完整的微服务系统需要先有一个框，然后再解构原来单体的功能，最后再部署微服务，这是一个有先后顺序的过程。

2.3 微服务的高级进阶

所谓高级进阶，其实是微服务设计模式的总结，而所谓的设计模式简单来说就是前人留下的一些设计经验。

2.3.1 得 API 者得天下

亚马逊的缔造者杰夫·贝佐斯 2002 年在给内部员工的一封信里提到了 4 条内容，就是利用这简单的 4 条规定，经过 13 年的发展，贝佐斯除了做了一个网上商城之外，还把他的云、API 管理做成了一个价值 50 亿美元的生意。

第一，无论是外部使用者还是内部使用者，我们企业内部所有的数据和功能只能通过 API 提供给使用者；第二，提供的 API 不仅要考虑与现在系统、其他的微服务、其他系统之间的协作，也要考虑便于公司外部调用；第三，除 API 之外，不允许有第二种方式连接部门间的系统；第四，以上规定如果不遵守，那么就要被开除。

从这个故事能看出前瞻性对于企业发展的重要性，如今可以不夸张地说“得 API 者得天下”。现在我们做的是企业内部和外部的交换，未来还会涉及行业的交换，即行业内会有标准的接口，其会以行业的标准提供给我们现在能提供的能力。

1. 为什么需要 API Gateway

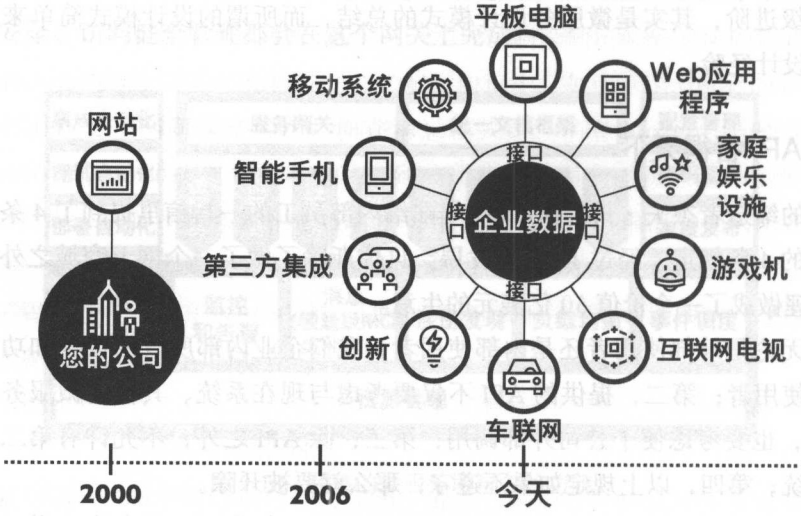
随着无线技术的发展和各种智能设备的兴起，互联网应用已经从单一的 Web 浏览器时代演进到以 API 驱动的无线优先（mobile first）和面向全渠道体验（omni-channel experience oriented）的时代，如图 2-19 所示。

PC 端的程序需要关注浏览器和分辨率；移动端的程序需要关注带宽和手机电量，甚至是地区的发展水平（也许使用短信登录更符合当地的实际）；在平板电脑上，我们很难使用右键操作。不同终端的操作特性带来了不同的要求，适配成为一个问题，我们可以通过将服务功能进行不同的组合，为 Web、移动端、可穿戴设备提供不同的体验。但如果组织一个移动端的页面，需要调用 20 个服务，这对于移动设备来说会有些吃力，这时我们可以使用 API Gateway 来缓解这一问题，这种模式下多个底层的调用会被聚合成一个调用。

使用 API Gateway 解决了两个问题：一个问题是如何让外部诸多请求直接找到我们提供的 API；另外一个问题是 API 之间的调用关系由谁来完成。由于互联网是全渠道的，因此除了外部直接访问这个问题需要解决，还要解决不同终端的适配。用户的接入形式多样，包括无线（手机、iPad）、Web、互联网电视、第三方合作应用等，各种用户设备的屏幕大小、操控体验方式不同。所以关于 API Gateway 定位做了两件比较重大的工作：一个是针对外部访问的健全、路由、控制；另一个是针对不同访问端页面的适配。简单来说，采用 API Gateway 需要：

- 封装内部系统的架构，并且为各个客户端提供 API。

❑ 理想情况下针对每种用户体验类型需要一个 API Gateway。



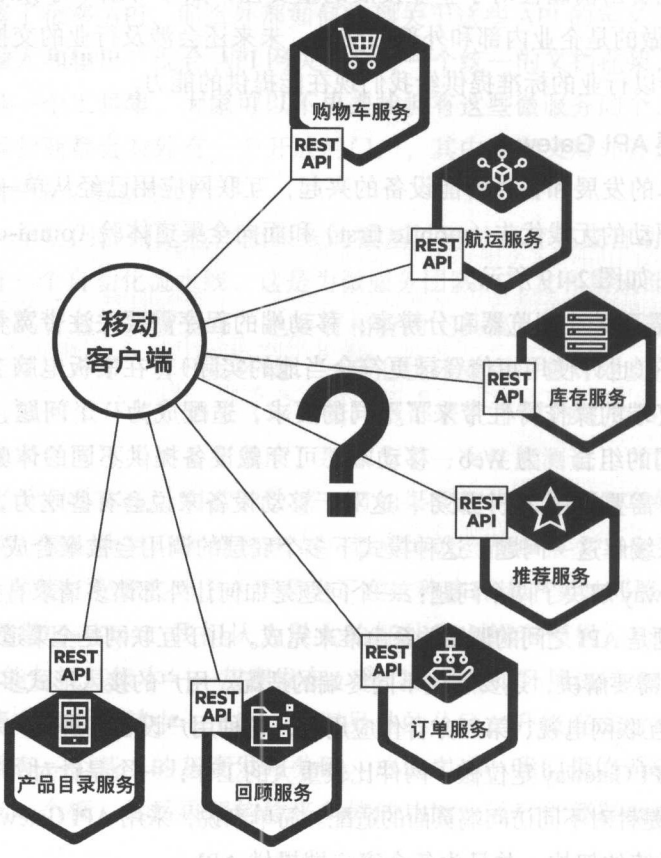


图 2-19 全渠道体验时代

❑ 还可能将授权、监控、负载均衡、缓存等功能封装到独立网关层。

但是任何事情都有两面性，有优势就有劣势。

❑ 优势：封装应用内部结构，减少和简化客户和服务器端的通信。

❑ 劣势：作为一个高可用的组件，必须需要开发、部署和管理，且可能成为开发瓶颈。

这也就是说使用 Gateway 的优势是解决了所有外部访问的问题，但劣势是一旦 Gateway 做的事情较多，就需要考虑它的性能要求，对资源的依赖度，出现单点问题如何解决等。

2. 如何实现一个 API Gateway

（1）性能和可扩展性

API Gateway 的性能和可扩展性非常重要。因此，创建一个支持同步、非阻塞 I/O 的 API Gateway 很有意义。目前已经有各类不同技术可以实现一个可扩展的 API Gateway。在 JVM 上，采用基于 NIO 技术的框架，如 Netty、Vertx、Spring Reactor、JBoss Undertow。一个可选的方案是 NGINX Plus，NGINX Plus 提供一个成熟的、可扩展的、高性能的 Web 服务器和反向代理，容易部署、配置和进行二次开发。NGINX Plus 可以管理授权、权限控制、负载均衡、缓存，并提供应用健康检查和监控。

（2）采用反应性编程模型

对于有些请求，API Gateway 可以通过直接路由请求到对应的后端服务上的方式来处理。对于另外一些请求，API Gateway 需要调用多个后端服务并合并结果来处理。对于诸如产品最终页面的请求，发给后端服务的请求是相互独立的，为了最小化响应时间，API Gateway 应该并发地处理相互独立的请求，但有时请求之间是有依赖的，API Gateway 可能需要先通过授权服务来验证请求，然后再路由到后端服务。利用传统的同步回调方法来实现 API 合并的代码会有回调函数麻烦，代码难以维护，反应性编程模式是很好的解决方案。

（3）服务调用

基于微服务的应用是分布式系统，并且必须采用线程间通信的机制。线程间的通信方法有两种：一种是采用同步机制，是基于消息的方法，这类实现方法有 JMS 和 AMQP，另外诸如 Zeromq 属于服务间的直接通信；还有一种线程间的通信采用异步机制，如 Thrift 和 HTTP。事实上一个系统会同时采用同步和异步两种机制。由于它的实现方式有很多种，因此 API Gateway 就需要支持多种通信方式。

（4）服务发现

API Gateway 需要知道每一个微服务的 IP 和端口，在传统应用中可能会硬编码这些地址，但在目前云基础的微服务应用中，这是一个非常简单的问题。基础服务通常会采用静态地址，可以采用操作系统环境变量来指定，但探测应用服务的地址就没那么容易了，应用服务通常动态分配地址和端口。同样地，由于扩展或者升级，服务的实例也会动态地改变。因此，API Gateway 采用系统的服务发现机制，要么服务端发现，要么客户端发现。

（5）处理部分失败

在实现 API Gateway 的过程中，另外一个需要考虑的问题就是部分失败。这个问题发生在分布式系统中当一个服务调用另外一个服务超时或者不可用的情况下。API Gateway 不应该被阻断并处于无限期等待下游服务的状态，但如何处理这种依赖于特定的场景和具体服务的失败？例如，如果产品详情页的推荐服务模块无响应，那么 API Gateway 应该返回剩下的其他信息给用户，因为这些信息也是有用的，推荐部分可以返回空，也可以返回固定的顶部 10 个给用户。但是，如果是产品信息服务无响应，那么 API Gateway 就应该给客户端返回一个错误。

在缓存有效的时候，API Gateway 应该能够返回缓存。例如，由于产品价格变化并不频繁，API Gateway 在价格服务不可用时应该返回缓存中的数值。这类数据可以由 API Gateway 自身来缓存，也可以由 Redis 或 Memcached 这类外部缓存实现。API Gateway 通过返回缓存数据或者默认数据，来确保系统错误不影响用户体验。

2.3.2 微服务的进程间通信

在单体式应用中，各个模块之间的调用是通过编程语言级别的方法或者函数来实现的。但是一个基于微服务的分布式应用是运行在多台机器上的。一般来说，每个服务实例都是一个进程。因此服务之间的交互必须通过进程间通信（IPC）来实现。

1. IPC 通信类型

当为某一个服务选择 IPC 时，首先需要考虑服务之间如何交互。客户端和服务器端之间有很多的交互模式，我们可以从两个维度进行归类。

第一个维度是这些交互是一对一还是一对多：

- ❑ 一对一：每个客户端请求有一个服务实例来响应。
- ❑ 一对多：每个客户端请求有多个服务实例来响应。

第二个维度是这些交互是同步还是异步：

- ❑ 同步模式：客户端请求需要服务器端即时响应，甚至可能由于等待而阻塞。
- ❑ 异步模式：客户端请求不会阻塞进程，服务器端的响应可以是非即时的。

表 2-2 显示了不同交互模式。

表 2-2 不同的交互模式

第二个维度 \ 第一个维度	一对一	一对多
同步	请求 / 响应	—
异步	通知请求 / 异步响应	发布 / 订阅发布 / 异步响应

一对一的交互模式有以下几种方式。

- 请求 / 响应：一个客户端向服务器端发起请求，等待响应。客户端期望此响应即时到达。在一个基于线程的应用中，等待过程可能造成线程阻塞。
- 通知（也就是常说的单向请求）：一个客户端请求发送到服务器端，但是并不期望服务器端响应。
- 请求 / 异步响应：客户端发送请求到服务器端，服务器端异步响应请求。客户端不会阻塞，而且被设计成默认响应，不会立刻到达。

一对多的交互模式有以下几种方式。

- 发布 / 订阅：客户端发布通知消息，被零个或者多个感兴趣的服务消费。
- 发布 / 异步响应：客户端发布请求消息，然后等待从感兴趣服务发回的响应。

每个服务都是以上这些模式的组合，对某些服务，一个 IPC 机制就足够了；而对另外一些服务则需要多种 IPC 机制组合。

了解了交互模式，接下来看看如何定义 API。API 是服务器端和客户端之间的契约，不管选择了什么样的 IPC 机制，重要的是使用某种交互式定义语言（IDL）来精确定义一个服务的 API。在开发之前，需要先定义服务的接口，并与客户端开发者详细讨论并确认。这样的讨论和设计会大幅度提高 API 的可用度以及满意度。API 定义实质上依赖于选择哪种 IPC。如果使用消息机制，API 则由消息频道（channel）和消息类型构成；如果使用 HTTP 机制，API 则由 URL 和请求、响应格式构成。

2. IPC 技术

目前有很多不同的 IPC 技术，服务之间的通信可以使用同步的请求 / 响应模式，如基于 HTTP 的 REST 或者 Thrift。另外，也可以选择异步的、基于消息的通信模式，如 AMQP 或者 STOMP。

对于同步方式而言，我们可以用请求 / 响应来概括整个过程，发起方发起一个远程调用后，发起方会阻塞自己并等待整个操作的完成，同步对于响应的低延迟有高要求，因而在目前这也是不太实际的。

异步的通信模型有两种。一种是基于消息的请求 / 响应方式，这与同步的不同，当使用基于异步交换消息的进程通信方式时，一个客户端通过向服务器端发送消息提交请求。如果服务器端需要回复，则会发送另外一个独立的消息给客户端。因为通信是异步的，客户端不会因为等待而阻塞，相反客户端理所当然地认为响应不会立刻接收到。

另外一种是基于事件的方式，服务提供方不发起请求，而是发布一个事件，然后期待调用方接收消息，并知道该怎么做，服务提供方不需要知道什么会对此做出响应，这也意味着可以在不影响服务提供方的情况下对该事件添加新的订阅。

使用消息机制有很多优点：

- 解耦客户端和服务器端：客户端只需要将消息发送到正确的 channel。客户端完全不需要了解具体的服务实例，更不需要一个发现机制来确定服务实例的位置。
- Message Buffering：在一个同步请求 / 响应协议中，如 HTTP，所有的客户端和服务器端必须在交互期间保持可用。而在消息模式中，消息 broker（代理）将所有写入 channel 的消息按照队列方式管理，直到被消费者处理。也就是说，在线商店可以接受客户订单，即使下单系统很慢或者不可用，只要保持下单消息进入队列就好了。
- 弹性客户端 – 服务器端交互：消息机制支持以上说的所有交互模式。
- 直接进程间通信：基于 RPC 机制，试图唤醒远程服务看起来跟唤醒本地服务一样。

然而，消息机制也有自己的缺点：

- 额外的操作复杂性：消息系统需要单独安装、配置和部署。消息 broker 必须高可用，否则系统的可靠性将会受到影响。
- 实现基于请求 / 响应交互模式的复杂性：请求 / 响应交互模式需要完成额外的工作。每个请求消息必须包含一个回复渠道 ID 和相关 ID。服务器端发送一个包含相关 ID 的响应消息到 channel 中，使用相关 ID 来将响应对应到发出请求的客户端。这个时候使用一个直接支持请求 / 响应的 IPC 机制会更容易些。

2.3.3 服务发现

在单体阶段，为了完成一次服务请求，代码需要知道服务实例的网络位置（IP 地址和端口）。传统应用都运行在物理硬件上，服务实例的网络位置都是相对固定的。例如，代码可以从一个经常变更的配置文件中读取网络位置。而对于一个现代的、基于云微服务的应用来说，这却是一个很麻烦的问题。服务实例的网络位置都是动态分配的，而且因为扩展、失效和升级等需求，服务实例会经常动态改变。因此，客户端代码需要使用一种更加复杂的服务发现机制。目前有两大类服务发现模式：客户端发现和服务器端发现。

如图 2-20 所示的客户端发现模式指当使用客户端发现模式时，客户端负责决定相应服务实例的网络位置，并且对请求实现负载均衡。客户端从一个服务注册表中查询，其中是所有可用服务实例的库。客户端使用负载均衡算法从多个服务实例中选择一个，然后发出请求。

客户端发现模式的优劣势如下。

- 优势：相对较为直接，不再考虑负载均衡，由于客户端知道可用的服务注册表信息，剩下的控制都可以由服务的请求端完成，可通过使用散列一致性更有效地实现负载均衡。
- 劣势：针对不同的语言注册不同的服务，在客户端需要实现每种语言的发现逻辑。

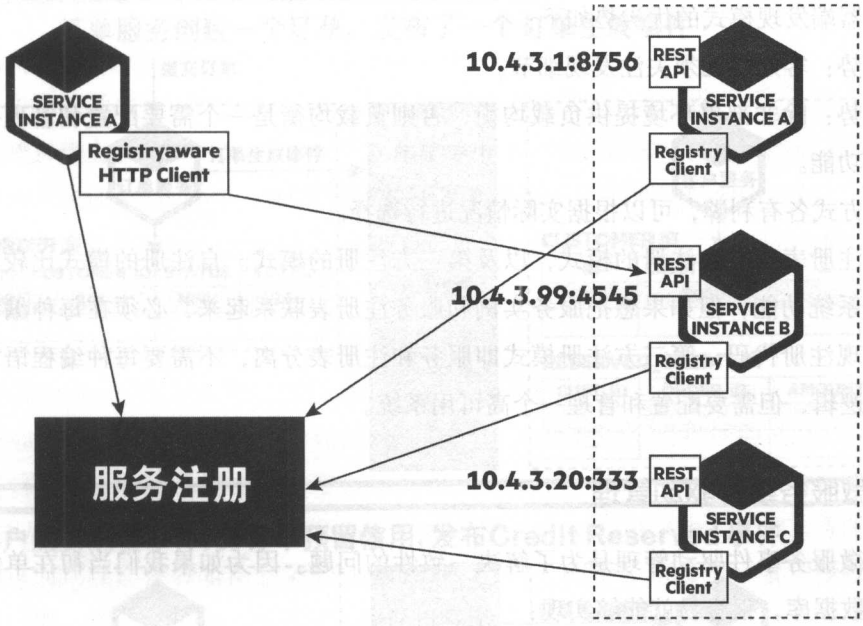


图 2-20 客户端服务发现

如图 2-21 所示，服务器端的发现模式指单独做一个负载均衡，这是与客户端发现模式最大的区别。还是通过注册表找到一个可用的服务，之后要把所有的请求发给作为服务器端的负载均衡，由服务器端的负载均衡具体地来找到可用的实例，然后进行相应的通信。

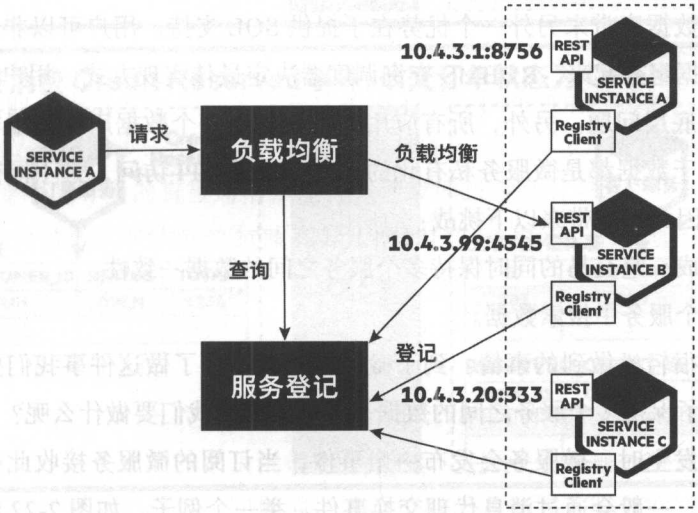


图 2-21 服务器端服务发现

服务器端发现模式的优劣势如下：

- 优势：客户端无须关注发现细节。
- 劣势：除非部署环境提供负载均衡，否则负载均衡是一个需要配置和管理的高可用的功能。

两种方式各有利弊，可以根据实际情况进行选择。

服务注册表也有自注册的模式，以及第三方注册的模式。自注册的模式比较简单，不需要其他系统功能，但如果想把服务实例和服务注册表联系起来，必须在每种编程语言和框架内实现注册代码。第三方注册模式即服务和注册表分离，不需要每种编程语言和架构完成注册逻辑，但需要配置和管理一个高可用系统。

2.3.4 微服务事件驱动管理

其实微服务事件驱动管理是为了解决一致性的问题。因为如果我们当初在单体里用一个交易型数据库，很容易就能够实现：

- 原子性（任何改变都是原子的）。
- 一致性（数据库状态始终一致）。
- 隔离性（即使交易并发执行，看起来也是串行）。
- 交易一旦提交不可回滚。

鉴于以上特性，应用可以简化为：开始一个交易，改变（插入、删除、更新），然后提交这些交易。

使用关系型数据库带来另外一个优势在于提供 SQL 支持。用户可以非常容易地通过查询将多个表的数据组合起来，RDBMS 查询调度器决定最佳实现方式，用户不需要担心如如何访问数据库等底层问题。另外，所有应用的数据都在一个数据库中，很容易查询。对于微服务架构，由于数据都是微服务私有化的，只能通过 API 访问，且不同微服务经常使用不同的数据库，因此可能带来以下挑战：

- 如何在完成一笔交易的同时保持多个服务之间的数据一致性。
- 如何从多个服务中搜索数据。

原来数据库很轻松做到的事情，到了微服务时代，为了做这件事我们要额外地做很多事情，目的是为了保持多个服务之间的数据一致性，那么我们要做什么呢？

当重要事件发生时，微服务会发布一个事件，当订阅的微服务接收此事件时，就可以执行相应的操作，一般会通过消息代理交换事件。举一个例子，如图 2-22 所示，订单如果生成了，要扣减个人的信用，只有信用扣减成功，这个订单才真正成功。

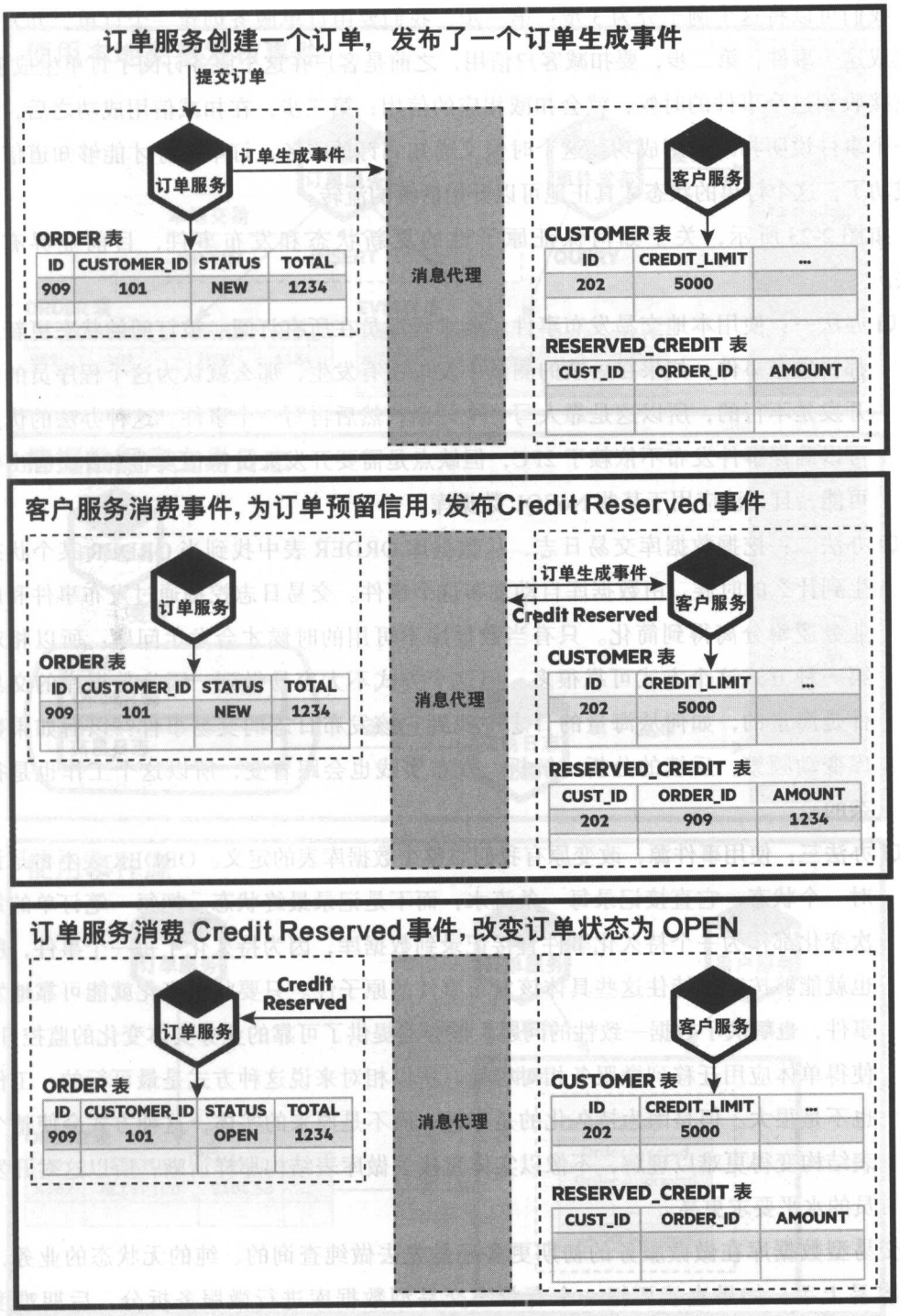

订单服务创建一个订单，发布了一个订单生成事件
提交订单
订单服务
订单生成事件
ORDER 表
ID CUSTOMER_ID STATUS TOTAL
909 101 NEW 1234
消息代理
客户服务
CUSTOMER 表
ID CREDIT_LIMIT ...
202 5000
RESERVED_CREDIT 表
CUST_ID ORDER_ID AMOUNT
客户服务消费事件，为订单预留信用，发布Credit Reserved 事件
订单服务
ORDER 表
ID CUSTOMER_ID STATUS TOTAL
909 101 NEW 1234
消息代理
订单生成事件
Credit Reserved
客户服务
CUSTOMER 表
ID CREDIT_LIMIT ...
202 5000
RESERVED_CREDIT 表
CUST_ID ORDER_ID AMOUNT
202 909 1234
订单服务消费 Credit Reserved 事件，改变订单状态为 OPEN
Credit Reserved
订单服务
ORDER 表
ID CUSTOMER_ID STATUS TOTAL
909 101 OPEN 1234
消息代理
客户服务
CUSTOMER 表
ID CREDIT_LIMIT ...
202 5000
RESERVED_CREDIT 表
CUST_ID ORDER_ID AMOUNT

图 2-22 事件驱动架构

我们可以将这个例子分为3步：第一步，我们要由订单服务创建一个订单，并发布订单生成这一事件；第二步，要扣减客户信用，之前是客户在这个服务订阅了订单生成事件，当它接收到这个事件的时候，就会扣减相应的信用；第三步，在扣减信用成功之后，再发布一个事件说明扣减信用成功，这个时候又通知到订单服务，订单服务才能够知道信用扣减成功了，这个订单的状态才真正地可以开始后续的流转。

如图2-23所示，关于如何保证原子性的更新状态和发布事件，目前业界有3种办法。

- 办法一：使用本地交易发布事件。要求程序员在所有订阅、被订阅的状态更新之后都要发布事件，如果在后续的测试中发现没有发生，那么就认为这个程序员的程序开发是不行的，所以这是靠人写一个订单，然后再写一个事件。这种办法的优点是可以确保事件发布不依赖于2PC，但缺点是需要开发人员牢记发布事件，有出错的可能，且无法应用于某些NoSQL数据库。
- 办法二：挖掘数据库交易日志，从数据库ORDER表中找到当ORDER某个状态发生到什么的时候，由数据库自动发布这个事件。交易日志挖掘通过发布事件和应用业务逻辑分离得到简化。只有当数据库不可用的时候才会发生问题，所以相对于第一种方式这个方式可靠很多。但这个方式不太容易做到，因为数据库的交易事件是海量的，如何从海量的日志中找到它该发布日志的交易事件？以后如果数据库变动频繁，后续的分析、解析、发布实践也会跟着变，所以这个工作也是很复杂的。
- 办法三：使用事件源，改变原有我们对整个数据库表的定义。ORDER表不再是订单对一个状态，它直接记录每一条流水，而不是记录最终状态。把每一笔订单的每一次变化都作为一个持久化事件直接记录到数据库，因为持久化了每一个事件，所以也就能够按时保持住这些具体该发布事件的原子性。只要状态变化就能可靠地发布事件，也解决了数据一致性的问题。该方法提供了可靠的业务实体变化的监控日志，使得单体应用迁移到微服务相对容易。所以相对来说这种方式是最可行的，工作量也不是很大。但是因为持久化的是事件，而不是原来的实体，这种方式会使整个库表结构变得更难以理解，不像以实体直接去做库表结构那样清晰，所以这对开发人员的水平要求更高。

交易型数据库在做微服务的初期更多还是先去做纯查询的、纯的无状态的业务，这样会容易上手。如果直接要把一个有状态交易型数据库进行微服务拆分，后期难度会更大。

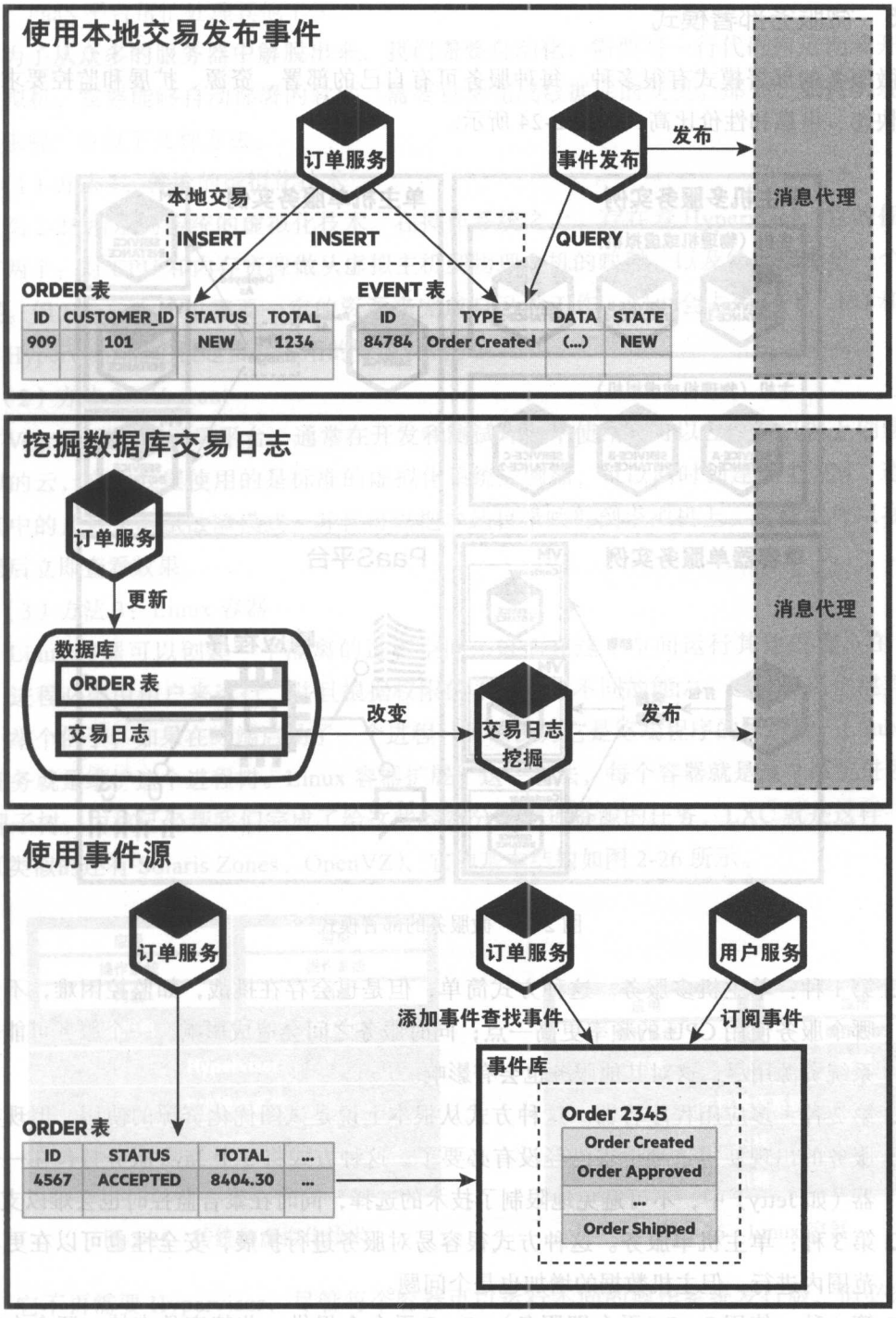


图 2-23 如何原子性地更新状态和发布事件

2.3.5 微服务部署模式

微服务的部署模式有很多种，每种服务可有自己的部署、资源、扩展和监控要求，但必须快速、可靠和性价比高，如图 2-24 所示。

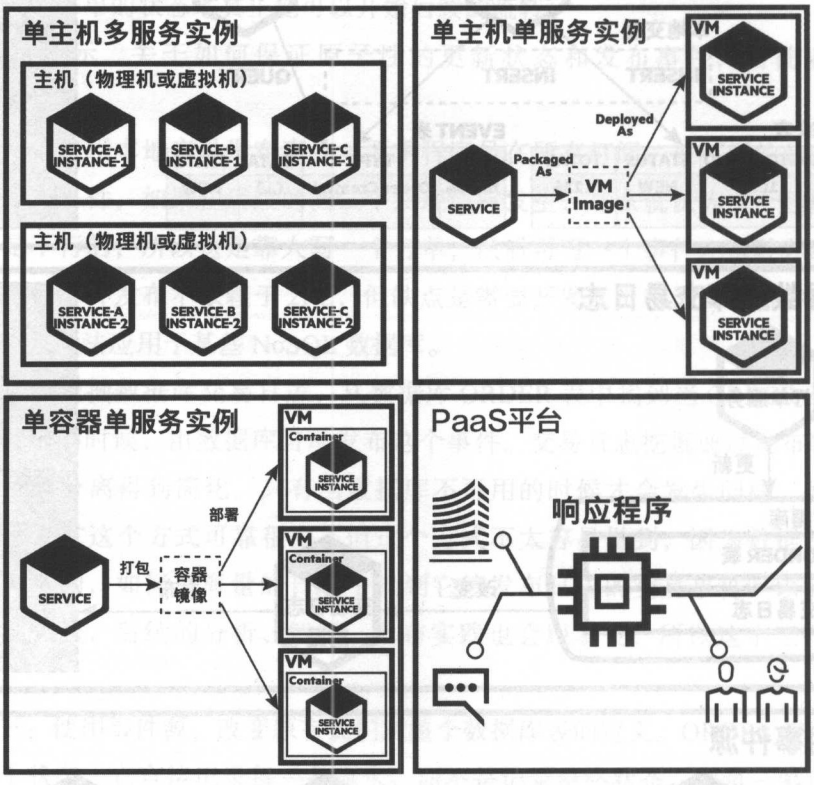


图 2-24 微服务的部署模式

- 第 1 种：单主机多服务。这种方式简单，但是也会存在挑战，如监控困难，不知道哪个服务使用 CPU 的频率更高一点；同时服务之间会造成影响，一个服务可能会使系统资源用尽，这对其他服务也会有影响。
- 第 2 种：多应用程序容器。这种方式从根本上说是试图优化资源的使用，但现在云服务的出现使得这种诉求已经没有必要了。这种方式将 5 个 Java 服务打包在一个容器（如 Jetty）中，不可避免地限制了技术的选择，同时在聚合监控时也会难以支持。
- 第 3 种：单主机单服务。这种方式很容易对服务进行扩展，安全性也可以在更小的范围内进行，但主机数据的增加也是个问题。
- 第 4 种：使用 PaaS（平台即服务）。PaaS 平台会提供一些特定的支持，还会自动配置机器然后运行，能够透明地对系统进行弹性管理，允许控制运行服务的节点数量，

PaaS 平台帮忙处理其他工作。

为了从众多的服务器中解脱出来，我们需要自动化，需要写一行代码来启动或开户一个虚拟机，需要能够自动部署的软件，需要自动完成数据库的变更。那么，如何实现上诉的需求呢？有以下几种方法。

（1）方法 1：传统的虚拟化技术

图 2-25 所示为传统的虚拟化技术。在操作系统之上，存在着 Hypervisor，它的任务主要有两个：对 CPU 和内存资源做从虚拟主机到物理主机的映射，以及给上层提供一个控制的层。但 Hypervisor 也需要一定的资源来完成自己的工作，它也会占用 CPU、I/O 和内存等，Hypervisor 的主机越多，占用的资源就越多。

（2）方法 2：Vegrant

Vegrant 是一个部署平台，通常在开发和测试环境中使用，可以在一台机器上创建一个虚拟的云，它的底层使用的是标准的虚拟化系统。例如，可以同时创建多个 VM，通过关掉其中的几台来测试故障模式，并且可以把本地目录映射到虚拟机上，这样就可以在修改代码后立即查看效果。

（3）方法 3：Linux 容器

Linux 容器可以创建一个隔离的进程空间，进而在这个空间运行其他进程。在 Linux 中，进程必须由用户来运行，并且根据权限的不同拥有不同的能力，进程可以创建其他进程。举个例子，如果在终端启动了一个进程，可以认为它是终端程序的子进程，Linux 内核的任务就是维护这个进程树。Linux 容器扩展了这一想法，每个容器就是整个系统进程树的一棵子树，内核已经帮我们完成了给这些容器分配物理资源的任务，LXC 就是这样一种容器（类似的还有 Solaris Zones、OpenVZ），它的基本结构如图 2-26 所示。

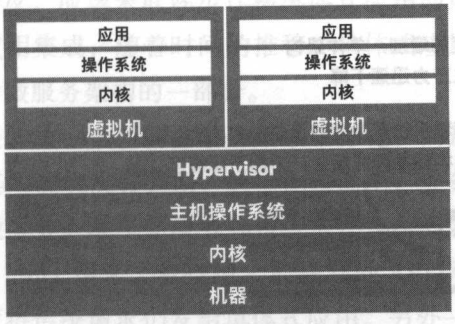


图 2-25 传统的虚拟化技术

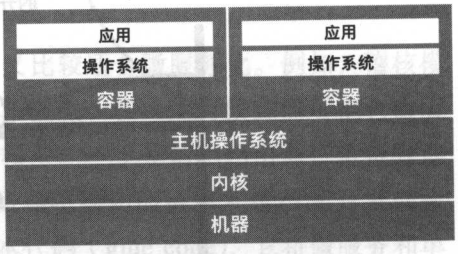


图 2-26 Linux 容器

它不再需要 Hypervisor，尽管每个容器可以运行不同的操作系统发行版，但必须共享相同的内核，因为进程树存在于内核中，这意味着我们的主机操作系统可以是 Ubuntu，而

在容器中可以运行 CentOS，只要它们的内核相同即可。

由于容器更轻量，所以在相同的硬件上能够运行的容器数量比虚拟机要多得多，而且启动速度更快，但容器在隔离性上还存在一定的问题。

（4）方法 4：Docker

Docker 是构建在轻量级容器之上的平台，可以处理大多数与容器管理相关的事情，我们可以在 Docker 中创建和部署应用，这些基于容器的应用与 VM 镜像类似，Docker 也能管理容器的配置，并处理一些网络问题。Docker 本身并不能解决所有的问题，它只是一个在单机上运行的简单的 PaaS，还需要一些工具来管理多台机器上的 Docker 实例上的服务。例如，当向这些工具请求一个容器时，它会找到容器并运行它。谷歌的 Kubernetes 和 Deis 就是这样的软件。

“Docker+ 调度工具”构成的解决方案介于 IaaS 和 PaaS 之间，我们可以称之为 CaaS(容器即服务)。

2.4 如何从单体架构迁移到微服务

虽然微服务是近年很热门的架构选择，但什么时候该选择微服务架构是有一定前提的。

图 2-27 是马丁·福勒所绘制的，其中黑线指的是单体，灰线是微服务架构。实际上在最初各方面效能、效率、开发、迭代的成果都比较好，不过随着单体越来越臃肿，各方面效能降低，这时候微服务的优势才得以体现。任何时候都是单体优先，只有单体结构变得越来越庞大，效能降低，并满足以下 4 个条件的时候才考虑进行微服务化：

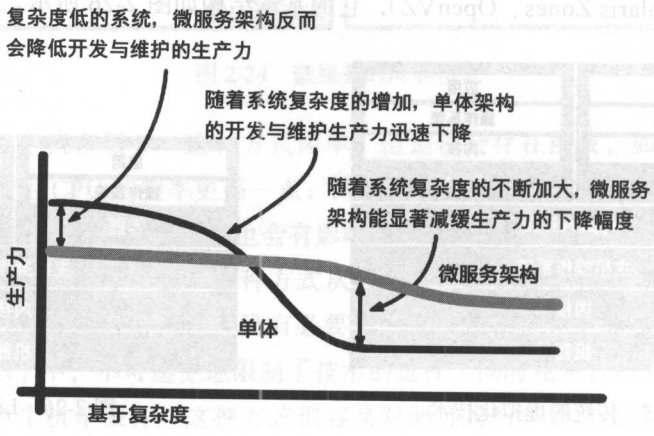


图 2-27 单体架构和微服务架构在不同系统复杂度下不同的生产力

❑ 要有快速迭代的能力。

- ❑ 要有基本的监控。
- ❑ 要有快速的集成。
- ❑ 要有一个 DevOps 文化。

图 2-27 表明了复杂度和生产率拐点的存在，但并没有量化复杂度的拐点到底是多少，或者换种说法，系统或代码库的规模具体达到多大才适合开始进行微服务化的拆分。在一篇有趣的文章《程序员职业生涯中的 Norris 常数》中提到，大部分普通程序员成长生涯的瓶颈在两万行代码左右。每一个瓶颈点的突破意味着需要新的技能和技巧，微服务合适的拆分拐点可能就在两万行代码规模附近，而每个微服务的规模大小最好能控制在一个普通程序员的舒适维护区范围内。一个受过职业训练的普通程序员就像一个拿到驾照的司机，一般司机都能轻松驾驭 100 公里左右的时速，但很少有能轻松驾驭 200 公里或以上时速的司机，即使能够驾驶，风险也是很高的，而能开“喷气式飞机的飞行员级别”的程序员恐怕在大部分的团队里一个也没有。

另外一个实施前提是基础设施的自动化，把 1 个应用进程部署到 1 台主机，部署复杂度是 $1 \times 1 = 1$，若应用规模需要部署 200 台主机，那么部署复杂度是 $1 \times 200 = 200$。把 1 个应用进程拆分成 50 个微服务进程，则部署复杂度变成了 $50 \times 200 = 10\ 000$，缺乏自动化设施，仅部署就是一件“疯狂”的事情。所以前面微服务的特征才有基础设施自动化，这与规模有关，也是因为其运维复杂度呈乘数级飙升，从开发之后的构建、测试、部署都需要一个高度自动化的环境来支撑，这样才能有效降低边际成本。

微服务化的原则是：要渐进改革，不要推倒重来。不要大规模重写代码，重写代码听起来很不错，但实际上充满了风险，最终可能会失败，就如马丁·福勒所说：“the only thing a Big Bang rewrite guarantees is a Big Bang!”

相反，应该采取逐步迁移单体式应用的策略，通过逐步生成微服务新应用，与旧的单体式应用集成，随着时间的推移，单体式应用在整个架构中的比例逐渐下降，直到消失或者成为微服务架构的一部分。

最好从一个新的需求开始微服务化，而这个需求又比较适合微服务化。例如，稽核模块和其他模块没有关联，通信比较简单，业务逻辑比较独立，在这个时候我们可把它做成一个微服务。除了新服务和传统应用，还要新增两个模块。一个是请求路由器，负责处理入口（http）请求，有点像之前提到的 API 网关。路由器将新功能请求发送给新开发的服务，而将传统请求仍发给单体式应用。另外一个是胶水代码（glue code），它将微服务和单体应用集成起来，微服务很少能独立存在，经常会访问单体应用的数据。胶水代码可能在单体应用或者微服务或者两者兼而有之，负责数据整合。微服务通过胶水代码从单体应用中读写数据。

微服务可以通过 3 种方式访问单体应用数据：

- 访问单体应用提供的远程 API。
- 直接访问单体应用数据库。
- 自己维护一份从单体应用中同步的数据。

胶水代码也被称为容灾层（anti-corruption layer），这是因为胶水代码保护微服务全新域模型免受传统单体应用域模型的污染。胶水代码在这两种模型间提供翻译功能。开发容灾层可能不是很重要，但却是避免单体式泥潭的必要部分。

将新功能以轻量级微服务方式实现有很多优点，如可以阻止单体应用变得更加无法管理。微服务本身可以开发、部署和独立扩展。采用微服务架构会给开发者带来不同的切身感受。然而，这种方法并不解决任何单体式本身问题，为了解决单体式本身问题必须深入单体应用做出改变。下面我们来看看这么做的策略。

（1）排序模块转成微服务

一个巨大的复杂单体应用由上百个模块构成，每个都是被抽取对象。决定第一个被抽取模块一般极具挑战，最好是从最容易抽取的模块开始，这会让开发者积累足够的经验，这些经验可以为后续模块化工作带来巨大的好处。转换模块成为微服务一般很耗费时间，可以根据获益程度来排序，一般从经常变化的模块开始会获益最大。一旦转换一个模块为微服务，就可以将其开发部署成独立模块，从而加速开发进程。

将资源消耗大户先抽取出来也是排序标准之一。例如，将内存数据库抽取出来成为一个微服务会非常有用，可以将其部署在大内存主机上。同样地，将对计算资源很敏感的算法应用抽取出来也是非常有益的，这种服务可以被部署在有很多 CPU 的主机上。通过将资源消耗模块转换成微服务，可以使得应用易于扩展。

查找现有粗粒度边界来决定哪个模块应该被抽取，也是很有益的，这使得移植工作更容易和简单。例如，只与其他应用异步消息同步的模块就是一个明显的边界，可以很简单、很容易地将其转换为微服务。

（2）从哪里开始拆分：接缝

从接缝处可以抽取相对独立的一部分代码，对这部分代码的修改不会影响系统的其他部分，这些接缝就可以作为服务的边界。那么如何识别出接缝呢？我们可以使用前面所提到的限界上下文，可以通过程序中的命名空间，也可以通过工具来帮助我们，如利用 Structure 101 这样的工具来可视化包之间的依赖。

（3）如何抽取模块

抽取模块的第一步就是定义好模块和单体应用之间的粗粒度接口，由于单体应用需要微服务的数据，反之亦然，因此这更像是一个双向 API。因为必须在负责依赖关系和细粒度接口模式之间做好平衡，因此开发这种 API 具有挑战性，尤其对使用域模型模式的业务逻辑层来说更具有挑战。因此经常需要改变代码来解决依赖性问题，一旦完成粗粒度接口，

也就将此模块转换成独立微服务了。为了实现，必须写代码使得单体应用和微服务之间通过使用进程间通信（IPC）机制的 API 来交换信息。第二步迁移就是将模块转换成独立服务。内部和外部接口都使用基于 IPC 机制的代码，抽取完模块，也就可以开发、部署和扩展另外一个服务了，此服务独立于单体应用和其他服务。

可以从头写代码实现服务，在这种情况下，将服务和单体应用进行整合的 API 代码成为容灾层，在两种域模型之间进行翻译工作。每抽取一个服务，就朝着微服务方向前进一步。随着时间的推移，单体应用将会越来越简单，用户就可以增加更多独立的微服务了。

（4）杂乱依赖的根源：数据库

为什么这么说呢？因为在通常情况下，我们在业务层的代码已经通过分层组织到相应的包中了，但是只有数据库是共用的，数据库对所有的代码都允许访问，是一个巨大的 API。

对于同一张表被多个限界上下文使用的场景，我们应该如何处理？以下是一些处理的步骤和原则：

1）分清代码中对数据库进行读写的部分。我们需要厘清代码是如何访问数据库的，在什么地方读，在什么地方写，它们分别位于什么样的上下文中。

2）打破外键关系。对于表与表之间的外键关系，如果这两张表需要被拆分至两个微服务中，我们可能需要放弃外键关系，同时把这个约束关系放到代码中实现，可能还需要实现跨服务的一致性检查，或者实现周期性触发清理数据的任务。我们可以通过类似于 SchemeSpy 这样的工具来分析数据库表之间的依赖关系。

3）共享静态数据。例如，国家、部门之类的数据都是各个微服务之间经常使用的，这些数据的特征是不会经常变化，而且通用性高。这些数据在微服务划分之后该如何处理呢？

- 方法一：我们可以为每个微服务复制一份这样的数据，但这会导致数据的一致性问题。
- 方法二：把共享的数据放入代码之中，如放在属性文件中，或者简单地放在一个枚举中，但数据一致性问题仍然存在。
- 方法三：把这些静态数据放在一个单独的服务中。

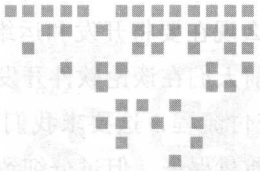
4）共享数据。如果不同的微服务都使用了同一张表，在这种情况下该如何分享？其实这种情况很常见：领域概念不是在代码中建模，相反是在数据库中隐式地进行建模。这里缺失的领域概念是客户，因而我们需要提供一个新的服务。

5）共享表。与共享数据不同的是，不同的微服务也会使用同一张表，但两者修改的部分不一样，在这样的情况下，我们可以把这张表拆分成两张表，分别供两个微服务使用。

6）实施拆分。通常，我们推荐先分离数据库结构，然后对代码进行拆分的方法。表结构分离之后，对于原先的某个动作而言，对数据库的访问次数可能会变多。这也是我们需要考虑的问题，这里涉及分布式事务的相关问题。

另外，先拆分数据库但不分离代码的好处在于，可以随时选择回退这些修改或是继续，而不影响服务的任何消费者。

总之，将现有应用迁移成微服务架构的现代化应用，不应该通过从头重写代码的方式实现；相反，应该通过逐步迁移的方式实现。在拆分的同时，需要同期配置服务治理平台，完成服务发现、配置管理、日志管理、监控等内容。平台和应用是可以考虑分开的，由团队专门负责。



第 3 章 Chapter 3

DevOps 实践

关于 DevOps 的由来，最早可以追溯到 2008 年的多伦多敏捷大会上，Andrew Clay Shafer 和 Patrick Debois 就“敏捷基础设施”进行了探讨。随后，2009 年在比利时根特召开的首届 DevOpsDays 活动上，Patrick Debois 首次在公开场合提出“DevOps”概念，“DevOps”随即成为席卷全球的热点话题，Patrick Debois 也被誉为“DevOps 之父”。2010 年在美国山景城（Mountain View）举办的 DevOpsDays 年会活动中，Damon Edwards 用一个缩写“CAMS”诠释了 DevOps，即文化（Culture）、自动化（Automation）、测量（Measurement 或 Metrics）和分享（Sharing）。随后 Jez Humble 将“L”（Lean，精益）原则也加入其中，最终变成了 CALMS。

“时代造就英雄。”在运维人员转型的迫切需求和容器这样的虚拟化技术快速发展的推动下，再加上互联网的时代背景，造就了 DevOps。

本章的 DevOps 实践内容主要从以下三方面帮助大家更好地了解 DevOps、实践 DevOps：

- 什么是 DevOps？DevOps 从哪里来？
- DevOps 实践的具体内容。
- 通过案例讲述如何将 DevOps 引入传统项目，又如何将交付的流水线用一些平台或工具搭建起来。

3.1 DevOps 思想导入

3.1.1 什么是 DevOps

究竟什么是 DevOps？

DevOps 是英文单词“ Development+Operations”的组合，从字面上看就是开发和运维的统一。为什么现在要将开发和运维两个部分的工作统一起来看呢？我们可以简单回顾一下，很多年以前人们在谈论软件开发模型及运维 ITIL 实践框架时，都从各自的角度强调做标准化的 IT 交付流程，这要求我们要针对软件交付的过程定义一些标准流程，才能使 IT 交付有稳定的质量保证。但过分细致的流程定义会带来 IT 交付团队组织架构的分割，会形成“开发是开发，测试是测试，运维是运维”这样条块分割的局面，实际上在传统的项目里也基本都是这么做的。而站在项目管理的角度一般会分成需求、开发、测试、运维等不同的组织结构，站在 IT 归属的企业角度则会将开发管理、运行管理和基础设施维护分成不同的组织结构，目的是让每一个团队的人更专业，专注在自己专业的领域里。但是由于组织结构划分带来各自出发点的不一致、各自 KPI 的不一致，并且导致在日常沟通过程中会出现更多争执，在严重的情况下，各部门已经不关心业务的状态和结果了，这完全背离了我们做 IT 系统的初衷。

DevOps 的核心思想是把所有 IT 交付和运维服务的团队统一起来，围绕着一个统一的业务价值目标及业务交付范围加强沟通，通过频繁、快速的迭代交付和反馈，达到缩短交付流程、加快交付速度和提升交付质量的目的。

传统 IT 项目的交付管理模式是瀑布模型，按照软件工程的生命周期过程一个步骤接着一个步骤地进行，先确定一个软件系统的功能范围，然后做开发和测试，接着上线部署，最后交给运维团队。随着 IT 系统的功能和架构越来越复杂，我们无法从整体上对所有 IT 功能和设计做详细的计划，原有的瀑布式项目管理模型已经行不通了，为了能适应如此复杂的需求变化，敏捷管理应运而生。在敏捷开发管理模式下，一般需要建立一个小规模的敏捷团队，拆掉需求、开发和测试的隔离墙，作为一个整体的交付团队紧密工作，从而以“小步快跑”的方式达到适应需求变化、缩短交付周期的目标。敏捷开发管理在一定程度上解决了开发交付的问题，但没有解决开发与运维分割对立的问题，因而还是不能从端到端解决软件交付的效率和质量问题。

DevOps 在需求和开发阶段就要考虑运维，包括程序运行需要的配置参数、基础设施参数、一级运维的便捷性等。如图 3-1 所示，DevOps 解决了很多痛点，而引爆 DevOps 的关键点是从运维的自动化部署开始的，从技术角度运维为了提高效率需要做自动化部署，这种需求在主机服务器的 X86 化、云化情况下更加强烈。当把 IT 基础设施从传统的几十台小机变成之后几百台 X86 服务器，甚至上千台虚拟机之后，如果在部署环节依靠人工部署，无疑是一场噩梦。一般在这种情况下，我们会不自觉地在部署环节想办法解决效率和质量问题，即如何把业务快速分发到生产上，让它们运转起来，自动化部署就这样应运而生了。在这种情况下，既然开发交付可以很自动地部署到生产上，那这种自动化的技术自然是可以向前延伸的，最后会延伸到自动化测试、自动化集成、代码编译的过程，贯穿起来就是

一条 DevOps 流水线。

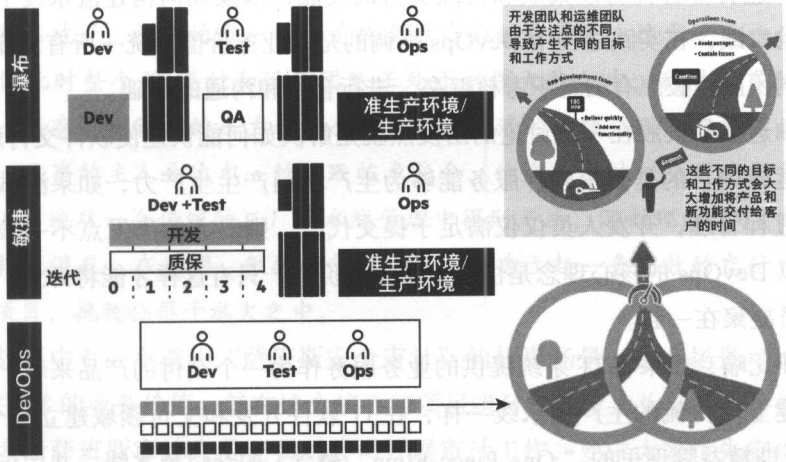


图 3-1 DevOps 解决的痛点

2009 年在比利时根特举办的首届 DevOpsDays 活动中，Patrick Debois 首次在公开场合提出“DevOps”这一名词。此后，“DevOps”随即成为全球 IT 界各种活动中热议和讨论的焦点话题。实际上 DevOps 不像 ITIL，ITIL 由英国政府部门 CCTA（Central Computing and Telecommunications Agency）在 20 世纪 80 年代末制定，现由英国商务部（Office of Government Commerce，OGC）负责管理，主要适用于 IT 服务管理（ITSM）。ITIL 为企业的 IT 服务管理实践提供了一个客观、严谨、可量化的标准和规范。也就是说，ITIL 实际上先出了一套标准，这个标准被国际组织所接受，然后慢慢地推广到企业中应用。但 DevOps 其实是来源于实践的，因此也没有对它做标准的定义。

目前有一些国际组织将 DevOps 进行了整理和定义：DevOps 强调对应用进行快速、小规模、可迭代的开发和部署，以更好地应对和满足客户需求。它要求进行文化的转变，即将开发和运维职能作为一个合作的整体来看待，关注于提供业务价值，主旨是精简整个 IT 价值链。

也可以这么说，DevOps 其实来源于敏捷管理（Agile Management）、持续交付、ITSM，也来源于精益管理（Lean Management）。

敏捷管理强调开发交付的过程，但没有讲运维如何控制；持续交付是开发的具体实践，即如何能快速地把交付的过程提交出去；ITSM 强调运营过程管控；精益管理强调管理的持续改进，缩短不必要的环节，避免浪费。对生产线过程要仔细分析，是否存在浪费，如果存在浪费就要把它拆减掉。所以我们会发现 ITSM 和持续交付分别从运维和开发角度讲述了我们实际应该怎么做，但是用精益和敏捷的思想可以把 ITSM 和持续交付结合起来，最后形成 DevOps。

那么 DevOps 具体来说到底是什么？我们认为 DevOps 是一套实践框架，包括了精益、敏捷的理念，包含了各种持续集成和持续交付的技能，以及如何构建流水线上的工具。它着眼于项目的实践，在实践过程中 DevOps 强调的是以业务价值来统一所有工作的目标，这个目标是不同团队打破原有的组织考核壁垒，进行合作和沟通的基础。

将开发和运维的人放在一起讨论的出发点就是解决如何能快速使软件交付形成生产力。写代码、做运营的目的是使得软件服务能够为生产部门产生生产力，如果测试人员仅仅立足于测试的过程管理，开发人员仅仅满足于提交代码，则双方的出发点不一致，肯定谈不到一起。所以 DevOps 的核心理念是快速实现业务价值，只有这样才能将开发、测试、维护以及需求人员凝聚在一起。

可以这样比喻，如果将 IT 系统提供的业务服务作为一个交付的产品来看，我们就是要像在工厂里建立单件流的生产流水线一样，在 IT 软件开发和交付领域建立一个 IT 服务交付的单件流（即精益管理里的“One Piece Flow”概念）的交付流水线，并用丰田精益的准时制（JIT）和质量检查（JKK）原则来建设这条流水线。

既然我们的目标是建立一个 IT 服务交付的单件流流水线，那就需要弄清楚以下几个问题。

- ❑ 这条流水线的内容是什么？它的起点在哪里？终点在哪里？
- ❑ 如何搭建这条流水线？
- ❑ 如何管理这条流水线？

带着这几个问题，我们来看一下 DevOps 的几个核心理念。

3.1.2 DevOps 核心理念

如果要将 DevOps 在项目中落实，我们需要关注的核心理念如图 3-2 所示。

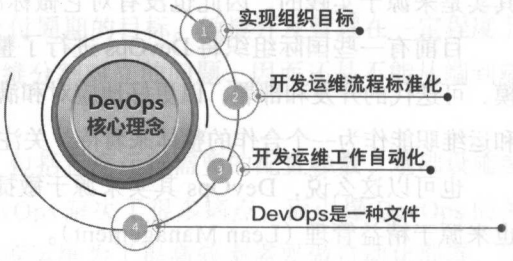


图 3-2 DevOps 核心理念

1. 实现组织目标，即业务价值

我们所做的软件系统是为业务部门的业务发展服务的，这是将所有 IT 交付团队统一起来的共同目标和原始驱动力。只要对比一下自己小团队的 KPI 和业务目标的关系就能发现，传统的分割式项目交付管理是多么浪费。实际上，仔细考虑了这个理念之后就很容易弄清楚 DevOps 流水线应该包含的内容，那就是整个开发、测试、部署和运维的过程都应该在这条流水线上，因为这些直接关系最终业务价值的实现，所以必须作为一个整体进行管理。

为了进一步阐述这个理念，我们可以看一下在《凤凰项目——一个IT运维的传奇故事》一书中的一个例子：

某公司濒临破产，挽救的唯一办法是改变原有的销售模式，这需要上一套先进的门店销售系统。但此时整个IT交付和运营团队正处于水深火热之中，种类繁多的项目、层出不穷的问题，还有安全审计的压力让整个团队处于崩溃的边缘，眼看着项目无法完成，公司即将破产。故事的主人公比尔·帕尔默临危受命，他在一个神秘公司的未来董事会成员的帮助下，不断地从一个传统的工厂流水线管理中得到灵感，深刻理解了IT交付的业务价值，让团队摆脱混乱，在透明、敏捷的管理文化下，建立起一套高效的交付流程，从而成功地交付了项目，挽救公司于水火之中。

在这个故事中有一个关于“萨班斯安全审计”的故事场景，这个场景可以进一步说明为什么要关注IT的业务价值。就在这个核心的项目组日以继夜地为新项目拼搏的时候，安全审计部门表示萨班斯审计人员正要对公司开展审计工作，要求大家优先完成配合，这种情况让项目组管理人员十分焦虑。比尔最终选择了将门店销售系统作为优先完成的工作，让项目团队更加专注，从而保证了关键的交付结果。事实上，萨班斯审计问题最终从非IT角度同样得到了圆满处理，这也告诉我们，还是要围绕业务来开展工作，而安全审计也会相应地变得更具弹性，而不再是死板的审计过程。

2. 开发运维流程标准化

在DevOps中并不是不讲流程，与之相反，在践行DevOps的时候需要的是标准化的交付流程，这个流程不是简单的管理规范，而是要用持续交付的流水线来取代开发运维流程标准化，实施这样的流程才会高效。

为什么DevOps要用持续交付的流水线管理来代替纯管理的流程定义呢？上文提到比尔的IT交付改善的灵感来自工厂里的产品生产流水线的管理过程。借鉴到IT领域，实际上就是将IT服务作为一个产品来交付，而为了提升这个交付过程的效率和质量，就需要参照工厂模式，搭建一条单件流的流水线，而这条单件流的流水线则需要一个标准化的流程来定义。

除了开发测试交付部分之外，从运维的角度来看，DevOps强调的是轻量化的ITSM流程和架构，即根据保证业务运行连续性的需要来裁减流程，并形成标准化的流程，所谓标准化指的是在需求、开发、测试、维护的过程中将流程最小化。流程过于复杂是造成IT资源浪费最主要的原因，所以一定要把流程最小化，将更多的精力、劳动、资源投入真正创造业务价值的生产中去。

3. 开发运维工作自动化

开发运维流程标准化是自动化的前提，如果流程不是标准化的，那么自动化也是没有根基的。只有将流程标准化，自动化才能有定义的标准。自动化会带来更多的好处，不仅

提升效率，让交付过程能够快速试错，同时还能带来过程效率和质量的透明化，让整个交付过程更加可控。

4. DevOps 是一种文化

DevOps 是一种文化，它倡导团队成员之间围绕共同的业务目标互相理解、信任和协作沟通，当交付过程出现问题后，更应该从中分析原因、吸取教训、增长经验，而不是互相指责和推卸责任。

总体来说，DevOps 可以用 CALMS 来表述，如图 3-3 所示，它是“Culture”“Automation”“Lean”“Measurement”“Sharing”这些单词组合而来的，代表的是自动化、精益、可衡量以及分享的文化。

图 3-3 CALMS

3.1.3 DevOps 术语

了解了 DevOps 的核心理念之后，我们需要掌握一些 DevOps 的术语，以帮助我们更好地理解 DevOps，如图 3-4 所示。

- 持续交付（Continuous Delivery，CD）：以小的增量块生产与交付完整软件的方法。
- 持续测试（Continuous Testing，CT）：自动化测试，包括单元测试、验收测试、功能测试、UI 测试、性能测试等。
- 持续集成（Continuous Integration，CI）：从开发人员写代码开始，到最后打包交付给环境（测试环境或类生产环境）结束，这个过程叫持续集成。要尽可能早地在开发周期中正确发现软件问题并确保整个平台的所有组成部分能够相互正确协作。

- 发布（Release）：单一代码包发布代码到测试、预生产、生产环境。
- 部署（Deploy）：推送版本到指定环境的行为。一体化过程中，在做架构时运维也参与其中，站在运维可运维性、安全性、可用性、监控需求的角度，提前审视需求架构是否合理和方便，能够快速、持续地反馈，也就是部署。

术语	是什么	不应该是什么
开发运维一体化	鼓励开发和运维团队以高度协作的方式朝着同一个目标共同努力的工作方式	开发和运维各自作战
持续交付	以小的增量块生产与交付完整软件的方法	每周或数月提供大块代码的方法
持续测试	包括单元测试、验收测试、功能测试、UI 测试、性能测试等	
持续集成	尽可能早地在开发周期中正确发现软件问题并确保整个平台的所有组成部分能够相互正确协作	问题被被忽略或跳过
发布	单一代码包发布代码到测试、预生产、生产环境	巨大的代码包交给运维处理
部署	推送版本到指定环境的行为	仅仅运维团队做的事情

图 3-4 DevOps 术语

3.2 DevOps 实践框架

从上面的表述中，基本可以看出 DevOps 其实不是一个简单的工具或者方法论，我们可以将 DevOps 看作一系列的实践论，包括项目敏捷管理的实践以及持续交付的实践等，既包括软件交付过程中的管理改进，也包括软件交付过程中的技术改进，关键是这些实践还需要结合起来统一推进。

因此，从项目中的实践来看，如图 3-5 所示，DevOps 是指导软件系统交付的系列实践方法，从项目计划、需求、设计、开发、部署、运维以及项目终止的整套过程中，DevOps 贯穿软件开发交付的过程。那么 DevOps 是如何贯穿的？我们站在传统 IT 项目的交付视角来看，DevOps 实践框架包括 4 个方面：敏捷管理、持续集成、持续交付和自动化测试。

- 敏捷管理：指将需求以用户故事的方式进行拆解，然后以最小化快速迭代方式进行开发管理。

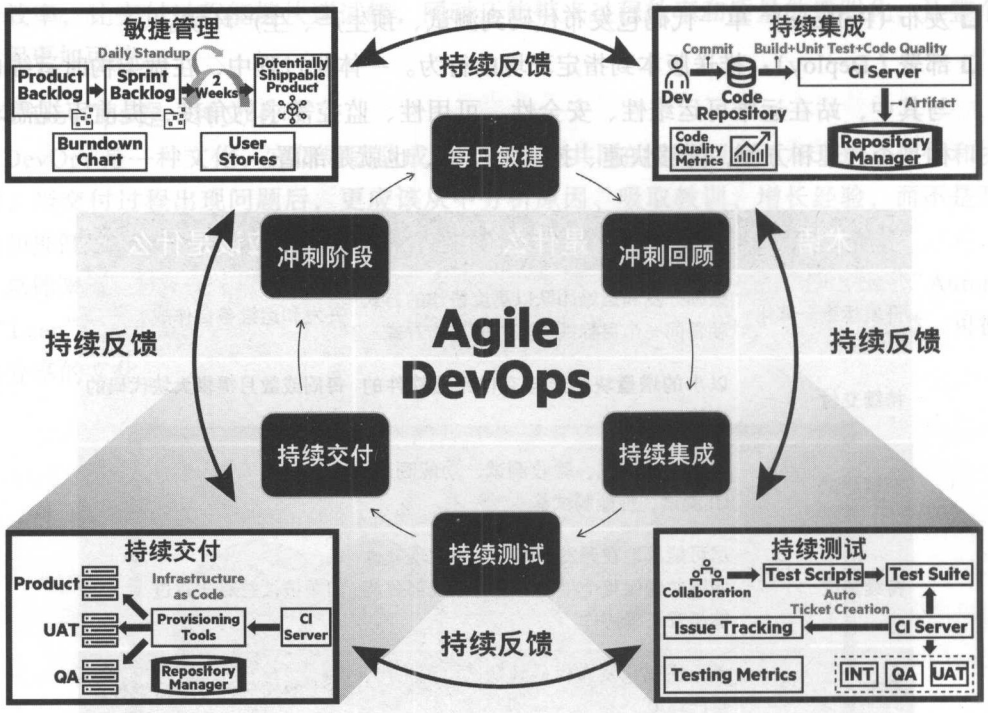


图 3-5 DevOps 实践框架

- 持续集成：指针对开发人员代码提交过程以单件流的方式进行流水线式自动化管理。
- 持续交付：指从写代码到生产过程是一条流水线，整个流程统一规划、预先定义，并以自动化、模板化的方式进行交付。
- 自动化测试：自动化测试在 DevOps 体系里实际上是持续集成的一个部分，如果是一个纯粹的 DevOps 实施项目，不应该将自动化测试拆出来单独考虑，但我们的目标是围绕传统的 IT 项目 DevOps 转型而言的，就需要团队直面过去的 IT 遗留资产，作为一种妥协的切入路径，我们建议可以将自动化测试部分单独拿出来看。但请读者千万不要认为自动化测试是可以在先期被忽略的部分，事实上，之所以将自动化测试单独拿出来说，正是因为在传统 IT 项目转型到 DevOps 的过程中，自动化测试正是最关键的部分。因为这部分的转变不仅涉及技术层面、工作习惯，甚至有组织转型方面的困难，所以也是 DevOps 转型过程中的“硬骨头”。

《凤凰项目——一个 IT 运维的传奇故事》一书阐述了整个 IT DevOps 借鉴的是工厂的生产流水线管理流程。从这本书的观念来看，我们会发现 IT 交付的过程其实是和丰田精益中 3 条管控流类似的一条流水线，也就是说 IT 的交付过程与工厂产品生产的流水线交付过程类似，管理改进的关注点是相同或者类似的，只是展现形式不一样。

既然我们已经很清楚 DevOps 的目标是搭建一条 IT 服务交付的单件流流水线，那么接下来就要考虑应该如何搭建这条流水线了，要回答这个问题我们需要学习一下丰田精益的几个核心理念。

在工厂流水线管理过程中，建立单件流的工作流程，降低产出在制品（Work in Process）是最关键的控制点。为了实现这个目标，需要清楚地定义单件流流程中各个环节之间的衔接关系，这就要用到精益里的 JKK（质量检查）思想。

工程生产过程中的精益理念需要最大化减少浪费，而减少浪费的关键是减少流水线上的在制品，或者叫半成品。而在制品问题在 IT 交付过程中同样存在并严重影响 IT 交付的效率，因此在搭建 DevOps 流水线之前需要分析一下项目交付过程中的价值流图，并找到价值流中的浪费环节，然后定义自己的单件流流水线。

JKK 源于丰田的一个思想理论，主要是在每一个环节里清楚地定义什么是“完工”，同时要求在每一个步骤里解决已经发现的所有问题，不要将已知的问题带入下一个环节。在 IT 交付过程中这一理论也表现得很明显。举个例子，在开发完成交付给测试的时候，可能会测试出很多 Bug，因为不太影响生产使用，勉强能用，就开始了部署。JKK 理论强调自己负责的工作不能在留有瑕疵的情况下交付给下一工作模块。如果是这样，就意味着后续的工作也将可能是无用功。在信任的前提下，每一个人在流水线上做好自己的本职工作。另外强调流程不要做得过于网状结构化，不要太复杂，这样会带来不可预知的资源瓶颈。流程过于复杂，将无法有效地调配资源，包括人、环境，以及影响生产力的任何资源。

持续交付体现的也是精益管理的思想，在当前成熟的互联网企业软件交付过程中，基本都可以做到持续交付，从写代码到生产整个过程基本无须人工干预。例如，腾讯开发人员写完代码之后，整个测试都是自动化的，交付流程是标准定义的，生产环境是纯粹的分布式架构。在部署的过程中根据地域考虑不同地方人群的集中度并进行自动化部署，之后还有一套自动化验证系统。也就是说，产品经理只需要关注开发，因为整个交付过程是自动化的流水线，所以代码写完了基本上就可以上线了。

IT 服务管理指的是传统 IT 管控方面的工作，DevOps 没有否定流程管理，但希望流程管理是轻量化、精简、最小级的流程，是精益管理的思路。我们要把现有流程中浪费资源、对生产力影响不大的环节精简或者优化。

以上是 DevOps 的实践理论框架，但是到具体项目里如何执行，我们总结了以下几个层次，如图 3-6 所示。

1）第一个是“道”。如果准备实施 DevOps，就要先找到所要交付的 IT 服务的业务价值点，要先找业务导向性明显的项目去落地 DevOps，这样团队会更清楚应该关注什么。例如，一些管理系统在项目里落地 DevOps 的时候，除非管理思路特别明确，否则不同团队之间在描述共同价值的时候并不太容易，团队合作的基础就会大大削弱。所以在选择 DevOps

试点项目的时候，要找到高业务价值的系统去做尝试，然后建立开发、交付、反馈的机制。换句话说，就是一定要把需求、开发、测试、运维这几个不同团队的人聚在一起，先弄清楚我们要做的 IT 系统是如何为业务服务的，支撑的业务是如何运营并为公司盈利的，这样团队才能找到沟通和交流的共同语言。

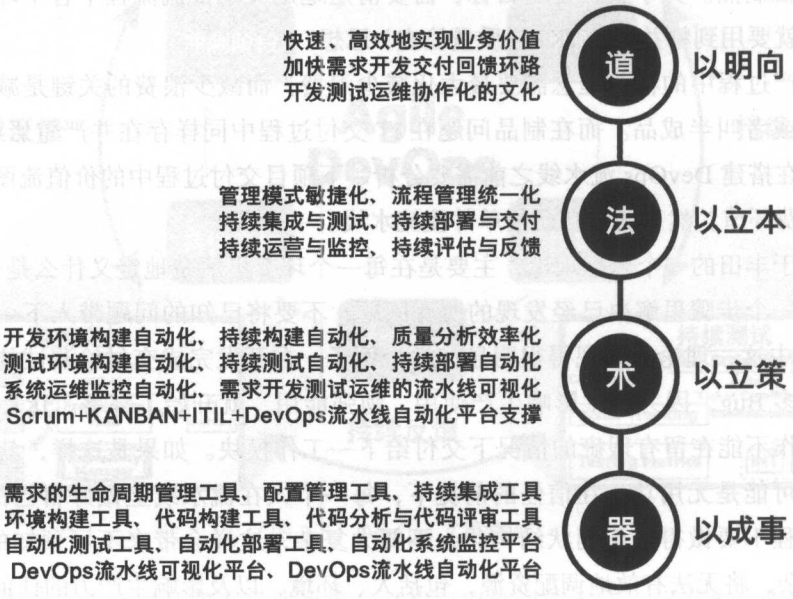


图 3-6 DevOps 的执行

2）第二个是“法”。即要引入什么策略来实现 DevOps 的实践，站在具体的策略上来说，应该从以下 6 个方面来考虑。

- 管理模式敏捷化：具体引入何种策略取决于管理模式的敏捷化，因为敏捷管理其实是 DevOps 最核心的思想。没有敏捷管理，单纯做 DevOps 工具是无法做好流水线的，原因就在于没有敏捷这种组织结构和文化来真正驱动 DevOps。
- 流程管理统一化：根据具体实施项目的背景和组织文化，对原有的 ITSM 流程进行精简和标准化。
- 持续集成与测试：搭建持续集成流水线很简单，但要搭建自动化测试框架并集成就比较困难了，因为传统项目基本上很难实现整个交付过程中自动化的测试。在互联网项目里，需求分析是以用户故事场景来驱动的，这样会带来一个好处，测试验收的案例是天然形成的，因为需求是根据业务使用的每一个步骤定义出来的。我们平时在网上使用支付宝，应用场景是先选购一个商品，然后打开支付页面，选定账号，提交订单，最后输入密码。每一步的场景定义都很清楚，这样可以很方便地做自动

化测试。对比来说，一个电信运营商的业务支撑系统的应用场景是一堆复用的场景，也就是说，语音套餐是一个过程，宽带如何复用或者互相影响，这个场景其实很复杂。一开始做需求分析很难把这个场景故事化，这样就没有办法搭建很全面的自动化测试使得代码真正流转过去。当然，DevOps 也不强调测试一定要全部自动化，通常来说我们希望单元测试、验收测试等需要在持续集成的流水线上自动执行，功能测试和性能测试等也应该尽可能地自动化实现。

- 持续部署与交付：即需要对所有环境进行统一的定义和管理，并在此基础上制订自动化交付的管理策略，并以自动化部署和验证的方式实施。
- 持续运营与监控：程序部署完成之后，还需要一系列自动化的方式来对业务运营情况进行管理和监控，如果程序设计足够服务化和弹性，那么就可以将运营监控的工作集中于业务的连续性监控，这是一种理想的运营模式，但是需要业务服务架构的支持。
- 持续评估与反馈：站在运营管理的角度，需要将生产系统的运行情况进行持续评估，并反馈到所有的项目成员那里。

3）第三个是“术”，即用什么方式来开展这方面的工作，要关注哪些具体的事情。这些事情包括开发环境构建自动化、持续构建自动化、质量分析效率化、测试环境构建自动化、持续测试自动化、持续部署自动化、系统运维监控自动化、需求开发测试运维的流水线可视化、Scrum+KANBAN+ITIL+DevOps 流水线自动化平台支撑等。

4）第四个是“器”，在 DevOps 成熟的开源社区里有很多工具都可以直接拿来用，这些工具可以帮助我们搭建自己的 DevOps 流水线。

3.2.1 敏捷管理

进入互联网时代常会提到 VUCA（Volatility，异变；Uncertainty，不确定；Complexity，复杂；Ambiguity，模糊），简单总结为“唯变不变”。企业要快速响应外部环境、市场变化以及组织发展的需要，要有敏捷的思想以顺应环境的需要，快速响应需求而不再拒绝需求。很多时候在传统的瀑布分阶段的模型中，从需求阶段开始，需要一次性设计大量的需求，再进入设计阶段，真正开始编码实施时已过去几个月，常常会出现开发到一半时需求变化的情况，项目团队会本能地抗拒。而敏捷的思想主张应该拥抱变化，采取与客户合作的方式来面对项目过程中的不确定性，以“小步快跑”的迭代方式快速交付软件，循序渐进，而不是一次性处理所有的需求，来降低项目实施过程中的不确定性和风险。

1. 引入敏捷管理，加速交付效率

微服务、云和 DevOps 三个课题是关联的，如果没有微服务，IT 系统架构在没有做云化和微服务化改造之前，做 DevOps 是很困难的，最大的困难就是敏捷团队无法拆分。因

为只有做了微服务化拆分，业务服务作为一个个独立的单元，才可以实现敏捷团队并行的开发、快速的交付。也就是说，当我们拿到一个需求时，这个需求只涉及其中某一个服务，无须与其他需求交叉，那么就可以独立开发测试，快速完成提交。但对于传统的即时架构，IT 系统没有办法做到这一点，不同的需求或功能往往调用同一个程序，在交付的时候是同一个交付，复杂系统的开发团队有 30 个人很正常，30 个人提交的代码会打在同一个包里，只能等 30 个人的代码都提交了，编译通过，才能实现交付。也就是说，30 个人必须同步，在这种情况下实践敏捷是“敏捷”不起来的。所以，我们尝试做 DevOps 实践，应该选择微服务化拆分做得比较好的系统来组建敏捷团队，引入敏捷的最佳实践“拥抱”用户需求的变化。

缩短交付周期，要以微服务化的系统框架为基础，尽可能地实践独立交付，通过自动化的编译、测试、部署，流水线快速上线，加速 DevOps 整个过程的流动效率。与此同时，在自动化过程中将质量的控制手段融入流水线中，当走到“最后一公里”，甚至到生产环境上去验证业务的质量时，如果程序已经微服务化了，那么在生产上的运行是一个负载均衡的架构。

举个简单的例子。例如，一个服务有 3 个 Docker 在给客户提供业务服务，在版本发布的时候可以先部署一个服务，然后引入一小部分用户的流量，例如，把 5% 的流量导入到新的版本上，在这个新版本上做冒烟测试来发现问题，从而快速得到反馈，大大减少新版本发布的风险，控制交付质量。以腾讯为例，他们有一整套业务验证机制，有很多测试的 QQ 号，有一套完整的场景验证、自动验证。当部署上去启动后，如所有资源都是真实的，就会在新部署的服务上提交业务。当然，在互联网领域也有可能把极少量真实的流量引入，没问题后再把剩下的流量引入，有问题直接回退也不会造成大规模的影响。在 DevOps 的最佳实践中有很多类似的方法可以实现在风险可控的情况下提高交付速度，所以在持续交付里有一个核心思想就是要保证历史的版本是可用的，是随时可以上生产环境的。DevOps 让一天成百上千次的部署成为可能，但这并不意味着发布了新版本，历史版本就可以扔掉了，在任何时候都要保证可以快速将有问题的版本退回到没问题的版本上去。在实际应用中要保留尽可能少的版本，毕竟支持多个版本是非常“痛苦”的。

如图 3-7 所示，总体来说，敏捷管理的核心思想首先是透明度。在快速交付的过程中，保证无论是通过工具、白板，还是通过系统使得从需求、开发、测试到上生产的过程，以及每一个需求在端到端流水线上都是可见的。例如，测试人员拿到一个需求的测试，在系统或者白板上能很清楚地看到其前置的工作做到什么程度，这样测试人员就能够提前准备自己的工作。每个人在团队角色中承担的任务对结果的作用也是可见的、真实可信的。

其次是可检验。所谓可检验就是在交付环节要定义清楚什么是完成的标准（Definition of Done，DoD），对交付结果的检验标准不能模棱两可。在敏捷做用户故事管理的时候，

Backlog 完成状态通过 0/1 来判断，必须 100% 完成工作才认为这个工作是真的完成了，即便完成了 90%，也不可以先凑合着将任务传递到下一个工作环节，这种模糊的状态控制在敏捷管理中是不被接受的，在持续交付的流水线上更是不可能实施的，开发团队在每个 Sprint 交付产品的功能上增量。这个增量必须是可用的，所以产品负责人可以选择立即发布它。每个增量都在之前所有增量（经过充分测试的可执行软件包）的基础之上进行，以此保证所有的增量都能工作。随着 Scrum 团队的成熟，我们预期完成的定义会更丰富，包含更严厉的标准来保证高质量，每一个环节的确认过程一定是可明确检验的过程。

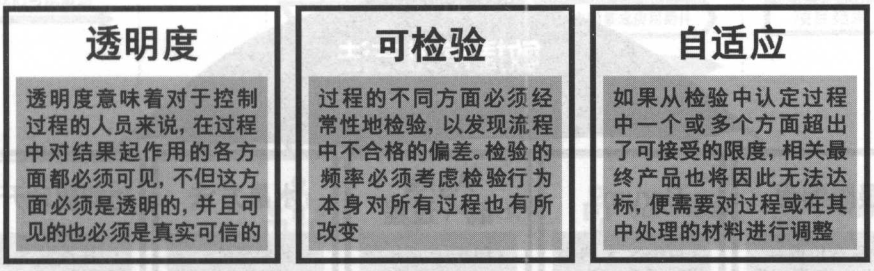


图 3-7 敏捷管理的核心思想

最后是自适应。当敏捷的过程出现偏离时，意味着最终产生的结果与目标不一致，不是用户所期望的或可接受的，要尽快调整，避免产生更严重的损失。例如，一个 Sprint 已经开始，团队成员正在开发中，此时客户提出重大变更会影响本轮 Sprint 的工作产品，作为产品经理应该尽快中止所有成员的工作，及时调整 Sprint 的范围，适应需求的变化，但中止 Sprint 这类事情是很少发生的。

2. 敏捷管理的基本框架

敏捷有很多模型，如图 3-8 所示，包括极限编程（XP）、Scrum、水晶（Crystal）、功能驱动开发（FDD）等，其中 Scrum 的普及率最高，并且融合到 DevOps 方法中。Scrum 有一个标准框架，包括角色、关键活动和工作产品。

Scrum 定义了 3 个角色。

- 产品经理：这是互联网领域的叫法，类似传统项目的需求人员。产品经理负责定义产品特性，编写用户故事，优先级识别、决定产品发布日期和内容，关注产品收益 ROI，接受和拒绝开发团队的工作产品。
- 敏捷教练：对项目组来说代表管理层，促进项目团队成员和产品经理的合作，及时为团队成员提供帮助。敏捷教练负责导入 Scrum 价值观念和最佳实践，确保每一个成员都认同 Scrum 价值观和遵守其游戏规则；清除对项目的外部干扰和障碍，确保团队功能完备且富有效率；促进所有角色和职能的紧密协作；Sprint 冲刺回顾会收集

团队成员的问题，并做流程的改善。

- 项目成员：一般 6 ～ 8 人，包括跨职能的角色、程序员、测试员、界面设计人员等。成员必须是全职投入的，团队自我组织，在理想情况下，团队成员是平等、不分头衔的，在一个 Sprint 中要保持成员稳定。项目成员负责将 Product Backlog 转化成 Sprint 中的工作项目，所有团队成员协调，合作完成 Sprint 中每一个规定的交付物。

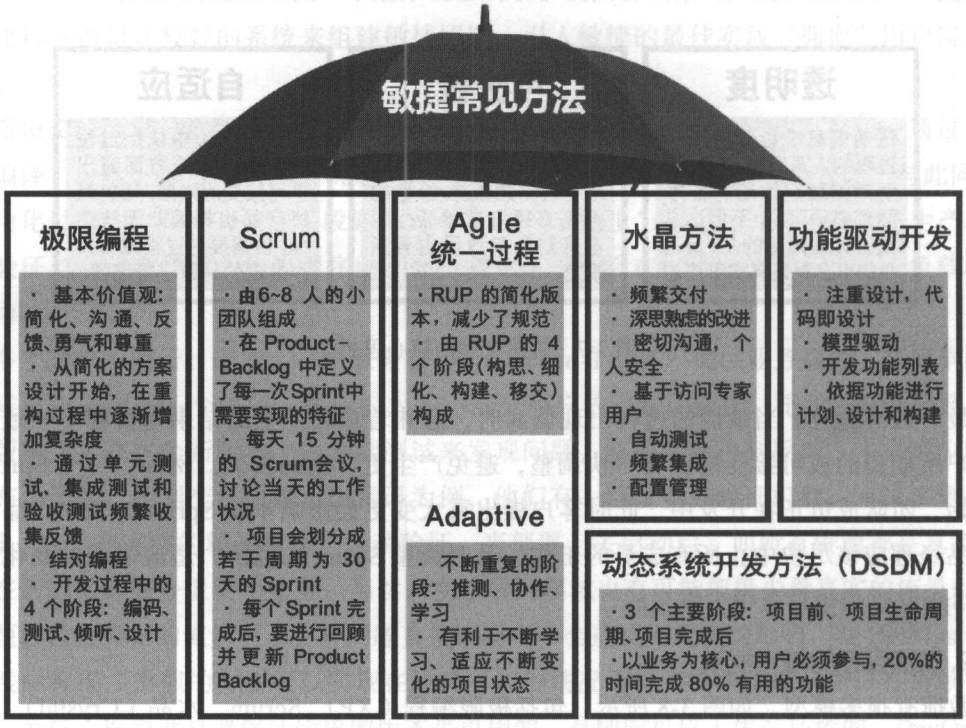


图 3-8 敏捷的常见方法

敏捷框架包括 4 个关键活动：冲刺规划会、每日站立会、冲刺复审会、冲刺回顾会。还有 3 个核心工作产品：产品订单（Product Backlog）、冲刺订单（Sprint Backlog）、燃尽图（跟踪过程中的状态图）。可以根据它们进行进度跟踪。

如图 3-9 所示，拿到需求以后首先要定一个版本（Sprint），版本交付周期有固定的时间箱（time box），一般是 2 ～ 4 周，在版本跟踪过程中最核心的是每日站立会（daily stand meeting）。版本交付完成之后，要针对版本的交付过程开一个总结会，总结问题和经验。冲刺回顾会即从管理角度看流程有什么要改变，通过回顾的方式持续改进流程，冲刺回顾最好在每次迭代时召开，回顾是敏捷过程真正的加速器。

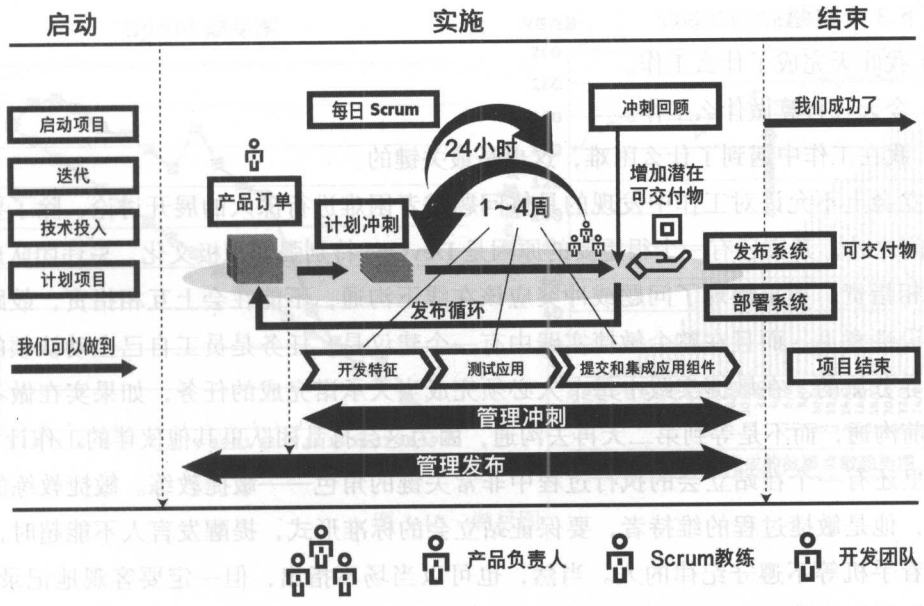


图 3-9 敏捷框架的 4 个关键活动

3. 每日站立会

真正效率的提升是通过站立会来具体实现的，形式上要求项目组在每天的固定时间、固定地点、固定 15 分钟组织站立会。为什么说站立会是核心？每日站立会又是如何开的呢？

首先，建议项目引入看板，通过看板来展示项目组每个人的任务和完成状态。现在有很多专门的电子看板，也有一些项目管理工具提供看板的功能，虽然电子看板的功能十分强大，但在导入敏捷实践初期还是建议大家用物理看板进行任务的跟踪。在站立会过程中，物理看板让项目组成员更清晰地聚焦各自的任务，在用户故事实施过程中的瓶颈也会被及时识别，等大家对这种沟通的形式熟练之后，可以将物理看板迁移至电子看板，提高效率。

关于如何开站立会，有以下几点建议提供给大家。

- 实际上在实施具体项目时，有的项目是早上开，有的是下班前开，但是还是建议尽量在早晨开。一般 15 分钟左右，而且一定是站着开，时间比较短，也有利于团队成员快速安排好当天的工作。
- 只有团队成员可以在站立会上发言，团队的领导可以参加这个会，但只建议旁听，不建议打断大家的发言。
- 每日 Scrum 站立会议由 Scrum Master 主持（初期），Scrum 团队所有成员轮流回答以

下 3 个问题：

1）我昨天完成了什么工作。

2）今天我打算做什么工作。

3）我在工作中遇到了什么困难，这也是最关键的。

站立会上不允许对工作中发现的具体问题或者困难进行深入的展开讨论，除了要缩短会议时间之外，其实还有一个很重要的原因是 DevOps 特别强调积极文化，忌讳团队成员之间的互相指责。如果出现了问题或冲突应该在线下沟通，不能在会上互相指责，鼓励大家提供建设性意见。而且在整个敏捷实践中有一个建议是，任务是员工自己主动认领的，而不是领导分派的。在敏捷实践中每个人必须完成当天承诺完成的任务，如果实在做不到也应该提前沟通，而不是等到第二天再去沟通，因为这会打乱团队里其他伙伴的工作计划。

这里还有一个在站立会的执行过程中非常关键的角色——敏捷教练。敏捷教练的工作很关键，他是敏捷过程的维持者，要保证站立会的标准形式，提醒发言人不能超时，提醒迟到、看手机等不遵守纪律的人。当然，也可以当场不指出，但一定要客观地记录下来，并且不做评价，做一个站立会“纪律榜单”，不守纪律的人都会上榜，会议结束将“纪律榜单”发给全项目组人看，恐怕就再也没有人愿意出现在这个榜单上。如此这般，站立会将顺利地开展下去。

4. 燃尽图

燃尽图通过可视化的方式展示进度，燃尽图用一条向下的曲线记录剩余工作，我们会发现任务是逐渐完成的，如图 3-10 所示。也就是说项目组的成员可以一目了然地看到整个项目的进展。如何保证项目的燃尽图是有效的呢？最关键也是最难的就是工作任务的分解。在 PMP 里有一个关键词——WBS（工作分解结构），PMP 建议每个任务包的计划工作时间不要超过 40 小时，也就是说每个任务的最大工作量是一个员工一周的工作量，而敏捷项目进度管理是按天来跟踪的。

举例来说，传统项目在周会上要总结一周计划的完成结果。如果超过 40 小时，需要两周完成，那么就无法评价当周工作的完成情况。而在敏捷项目里，用户故事（也就是需求）在冲刺计划中就被分解并细化到每个人的任务单元，而且每个任务的粒度最好小于 8 小时，每日站立会中跟踪每个任务单元的完成状态，任务管理的粒度越精细，会越早地发现与计划产生的偏差，在燃尽图中燃尽的曲线也会偏离资源计划曲线。所以，用户故事拆分粒度对燃尽图的影响很大，拆分粒度越小，则越能反映真实的状况，但也不是越小越好，否则拆分和沟通的成本也会增加。在整个敏捷管理过程中，如何让团队对工作的分解达到一个能够通过燃尽图跟踪的水平，工作任务的分解很关键，当然同时也是非常难做到的一件事情。

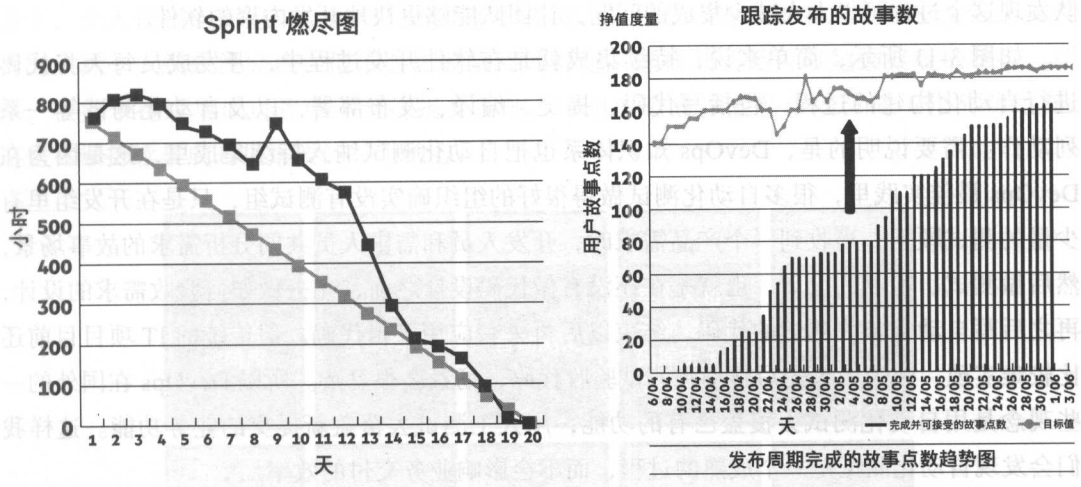


图 3-10 燃尽图

3.2.2 持续交付

持续交付是在原始需求识别到最终产品部署到生产环境的过程中，需求以小批量的形式在团队的各个角色（开发、测试等）间流动，以较短周期完成需求的小粒度频繁交付，并最终将发布产品给用户使用。

从代码开发和提交、资源管理、版本管理、自动构建到自动化测试、自动化部署等，再到最后形成持续反馈，都属于持续交付的范畴。持续交付的目标是尽快向用户交付有用的、可工作的软件，它强调过程的自动化、周期的快速迭代、结果的可用性和反馈，以及交付本身对团队成员的责任感。为创建一个软件发布的可重复、可靠过程，应坚持“将几乎所有事情自动化”的基本原则，把交付过程中涉及的所有环节都纳入自动化范围，如代码的构建和部署、功能测试和验收测试、数据库的升级 / 降级等，通过频繁地自动化提升交付效率，同时应将构建、部署、测试和发布的整个过程所需的东西都纳入版本控制管理之中，用版本控制为自动化设定质量基线，并以此来保障持续交付的顺利推进和落地。

3.2.3 持续集成

1. 什么是持续集成

什么是持续集成？持续集成是一种软件开发实践，即团队开发成员经常集成它们的工作，每个成员每天至少集成一次，也就意味着每天可能会发生多次集成。每次集成都通过自动化的构建（包括编译、发布、自动化测试）来验证，从而尽早地发现集成错误。许多团

队发现这个过程可以大大减少集成的问题，让团队能够更快地开发内聚的软件。

如图 3-11 所示，简单来说，持续集成就是在软件开发过程中，开发成员每天将代码进行自动化构建的过程，包括写代码、提交、编译、发布部署，以及自动化测试等一系列动作。需要说明的是，DevOps 知识体系也把自动化测试纳入持续集成里，这是因为在 DevOps 最佳实践里，很多自动化测试做得很好的组织确实没有测试组，只是在开发组里有少量的测试顾问，当收到一个产品需求时，开发人员和需求人员共同分析需求的故事场景，然后做测试、验收的设计，也就是在还没有做代码设计之前，先去做如何验收需求的设计，再之后写自动化验收测试的代码，结束以后再去写应用逻辑代码。但传统的 IT 项目目前还比较难实现，因为如果先写自动化测试验收代码，场景会很复杂。所以 DevOps 在国外的一些观念是用自动化测试去覆盖已有的功能，用人工测试去覆盖新提交的业务功能。这样我们会发现自动化测试是一个成熟的过程，而不会影响业务交付的效率。

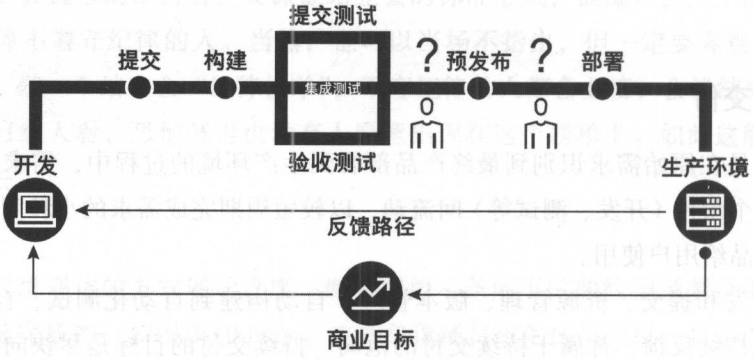


图 3-11　持续化集成流水线

2. 持续集成流水线搭建

持续集成的实施要求具有良好的团队纪律，团队成员可以严格执行持续集成过程中的各项规则，基于良好的配置管理，借助自动化工具创建并维护一个自动化构建和测试流程。

如图 3-12 所示，在持续集成过程中我们需要关注：

第一，代码提交是分布式提交，但是是由代码集中版本管理的，在这个过程中要确保代码的变更不会导致持续集成的失败。一个标准 DevOps 流水线中，其自动构建是每天自动完成的，这就要求在持续交付中确保自动构建的成功率。如果一旦自动打包失败，测试人员就无法正常开展工作，而开发人员首先要解决昨天的问题，这也就意味着敏捷交付又断线了。

举例来说，一个开发人员本来今天要提交一个代码，但下班时，在本地试着提交时结果报错。如果此时置之不理，肯定会导致最终打包失败，那么应该怎么办呢？第一个选择是把问题解决并成功提交；还有一种折中的选择，就是把今天的代码回退，在微服务的背

景下，个人代码的回退只会影响局部业务的提交，只是个人开发的新功能没有上线，这种情况虽然个人绩效没完成，但不会影响最终的打包，这样其实是可以勉强接受的状态。所以，在整个持续集成过程中不允许提交失败，一旦失败意味着整个团队的工作前功尽弃，这是在做持续集成时很重要的一个地方。

图 3-12 持续集成的要素

第二，开发人员每天至少向版本控制库提交一次代码。

第三，开发人员每天至少需要从版本控制库中更新一次代码到本地机器。

第四，需要有专门的集成服务器来执行构建，每天要执行多次构建。当然，有些传统项目在转型过程中实际构建已实属不易，在这种情况下最好的实践是每天晚上自动构建。

第五，每次构建都要 100% 通过。

第六，每次构建都要生成可发布的产品。

第七，修复失败的构建是优先级最高的事情。因为如果构建失败就意味着整个团队的工作计划会被打乱。

第八，测试是未来，未来是测试。

3. 持续集成工具的引入

（1）持续集成工具 Jenkins

Jenkins 诞生于 Hudson 项目，采用 Java 语言开发而成，能够在 WAR 文件下载完成之后直接使用，在持续集成过程中最核心的工具就是 Jenkins，它从始至终都贯穿于 DevOps 流水线中，帮助我们实现持续构建和测试，并可通过外部方式监控运行任务，提供可视化的图形界面供使用者监控交付全流程。

Jenkins 是目前 DevOps 社区里最活跃的持续集成软件，也是按照标准的 DevOps 持续

交付的方式在社区里发布的。Jenkins 大约每周会发布一个新版本，因此 DevOps 的版本变化特别快。但开源的软件有一个兼容性问题，因此在基于开源的生态构建流水线时一定要考虑兼容性。

总体来说，Jenkins 是典型的基于 DevOps 思想、交付思想，在互联网社区交付的软件。它本身是插件式的，主要任务是做流水线上的纽带，把整个 DevOps 所需要的环节集成进去。我们有代码管理软件，有开发管理软件，有自动化测试管理软件，也有部署管理软件，所有环节都有开源产品的支持，而这些产品都可以插到 Jenkins 流水线上来搭建 DevOps 流水线。

（2）代码管理工具 Git/Gitlab

SVN 是在传统项目里用得最多的代码管理软件，首先 SVN 是集中化的，代码集中管理；其次，SVN 基于 Windows，可用性非常好。开发人员不需要了解软件怎么用，建一个目录就知道代码在哪里，文档在哪里。但目前在互联网领域 GIT 是比较常用的持续构建工具，Git 有两个分支：一个是私有云解决方案 Gitlab；另一个是公有云解决方案 Github。诸如 Jenkins 这类开源社区的代码就在 Github 上，只需要在互联网上申请一个账号，就能将代码在互联网上进行管理。企业如果不愿意公开代码，可以基于私有云解决方案 Gitlab 搭建自己的代码管理服务器。

Gitlab 相对于 SVN 而言，结构更复杂。作为 Gitlab 的管理员要进行很多设置和配置。而作为一个初次接触 Gitlab 的开发人员，如果没有培训，可能会不太清楚自己的提交会带来什么影响。总体来说，Gitlab 的优势体现在以下几个方面。

首先是分布式管理。Git 分为服务区端和客户端。客户端分 3 个区，分别是写代码的工作区、暂存区和本地仓库，开发人员的开发是可以离线的，代码提交在个人的计算机上，进行本地管理。提交分两个阶段，首先提交到本地仓库，当开发人员的计算机连接到服务器上时，就会提交到服务器端。

其次是 Git 可以定义不同的代码分支。代码与代码之间的集成关系是在项目中预先定义的，在编译的时候如果分支与分支之间有冲突，是自动强制发现的。而且开发人员在本地可以模拟提交，如果提交的代码与别人提交的代码有冲突，在提交的时候就可以发现。在传统的 SVN 管理中，冲突只有在打包的时候才能发现。这样一来，开发人员的工作成了无用功，浪费了时间，也挫伤了开发人员的工作积极性。

所以从管理上来说，Git 强调的是开发人员一定要知道整个代码的分支规则，代码在提交之前就可以发现是否存问题，而非提交代码合并之后才发现问题，这样大大提高了代码提交的效率。

（3）静态代码扫描工具 SonarQube

在持续集成里还有一个重量级的开源工具 SonarQube，它可以提供全面代码扫描报告，

并可以集成到自动构建中去。代码提交到服务器上会自动进行扫描，这对于开发透明化来说是很重要的过程。过去按年度、按项目阶段来做工作，在 SonarQube 的支持下现在可以每天持续去做。如图 3-13 所示，SonarQube 是一个特别强大的工具，可以支持动态 / 静态扫描，可以代替人工评审。SonarQube 可以自动检测开发人员是否遵守开发人员代码编写规范，并且附带各种功能，包括代码重复检测、内存泄露风险检测、安全风险检测等。同时，SonarQube 支持不同版本的分割。如果一个传统项目的中间过程引入了 SonarQube，还需要做好心理准备，因为 SonarQube 是一个标准化的扫描，如果项目代码之前一直没有进行过扫描，开发人员也是“各自为政”，一定会检测出很多风险，出具报告会有很多需要改正的地方，可以说是“不忍直视”。所以为了打消开发人员的疑虑，一般可以先做一个版本的区隔。也就是说从现在开始扫描一个版本，保证增加的代码一定要遵循质量要求报告，历史报告慢慢分析修正，一次性将历史代码中的漏洞和错误全部修正不太现实。

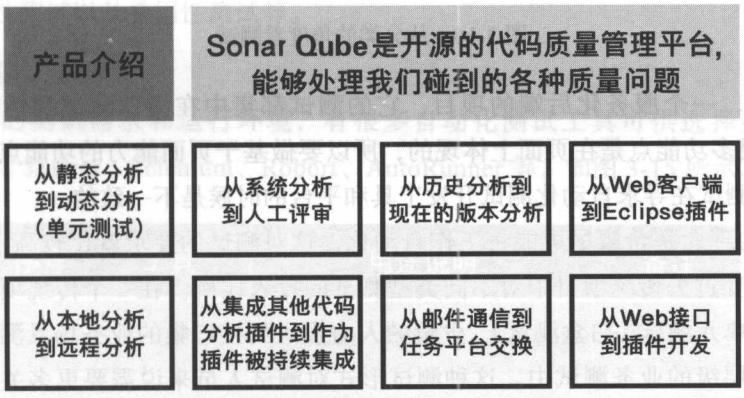


图 3-13 静态代码扫描工具——SonarQube

SonarQube 可以随时随地地介入代码交付过程。举例来说，SonarQube 有开发端的插件，当这个插件放到 Java 开发客户端时，我们会发现当写出来一个方法时它就会提出建议，指出问题所在，我们就可以现场修改。所以对于开发人员来说，如果前端就引入这个工具，即便增加代码也不会感到特别苦恼，因为一开始就会给出相应的问题提示，实现即时修改。开发人员在写代码的过程中就可以进行扫描，可以在本地提交以后在本地扫描，也可以在服务端代码收齐之后统一扫描，而且还可以在打包完成后扫描。作为一个工具，无论开发管理是什么流程，我们都可以把 SonarQube 放在大家可以接受的阶段引入进去。

3.2.4 持续测试（自动化测试）

1. 从不同视角看测试

测试是整个流水线上最耗工时、最需要投入的地方。在项目实践过程中，不同的项目

在自动化测试里最终很难形成一个统一的框架，因为测试要做的东西是不同类型的，有些系统是页面的，需要做页面测试；有些系统是松耦合架构的项目，需要关注功能端到端的流程；有些要关注性能、功能测试。如图 3-14 所示，不同的角度有不同的测试需求，针对这些不同的测试需求程序开发的语言也是五花八门，要想做自动化测试，这些因素必须都要考虑。

图 3-14 从分类的角度看测试

举例来说，一个服务化后端的项目，它的测试都集中在接口测试和性能测试上；而 CRM 项目有很多功能点是在页面上体现的，所以要做基于页面能力的功能点的测试。而接口测试和页面测试在寻求自动化测试开发工具和平台的时候是不一致的。

DevOps 的理念是希望尽可能将工作自动化，这对测试的要求就比较高了。从 DevOps 角度看需要改变过去传统项目中对不同类型测试的投入比例。在一个传统项目中，往往会忽略一开始的单元测试（白盒测试），少量的人力投入到接口集的业务场景测试，更多的人力会投入界面层级的业务测试中。这种测试形式对测试人员来说需要更多关注系统本身的功能，没有太多的测试技能和自动化技术技能方面的要求。但是关注系统功能的测试本身是有缺陷的，因为测试人员关注的是系统功能点，而不是业务本身，所以最后极有可能测试通过了，但到生产环境却不是客户想要的东西。

DevOps 希望从一开始就改变对测试的投入偏重，将测试分为 3 个层级。第一个层级是将最多的测试资源投入到白盒的单元测试中，同时加强代码的扫描和分析，在这个层面投入基本上没有太多的业务场景的分析，只需要关注技术规范的实现即可。在此之后的第二层关注业务场景的验收测试，这个层面的验收测试包括两个方面：一个是从业务故事场景设计测试案例，主要关注业务部门对业务场景的验收，保证提供的功能是能满足业务需要的；另一个是验收测试建议是从 IT 服务的接口层面设计的，而不是从页面来设计的，这样可以从业务实现代价和用户业务需求的满足两个方面寻求一个合理的性价比。最后一层的测试是 UI 测试，主要关注页面交互、业务属性和操作者的交互功能，这部分虽然是页面的功能，但是也建议用自动化的方式来实现。

单元测试是在开发代码的过程中就要同步考虑的，一般来说，在构建代码的过程中就立即触发，要求代码覆盖率至少达到75%以上。验收测试实际上是针对需求端的用户故事来说的，但是无论是单元测试还是验收测试的自动化测试方法，都建议用测试方法中的Happy模式进行设计即可，重点验证客户感知的业务场景是否有正确的输出结果。UI的测试也需要自动化实现，当下也有很多针对UI的自动化测试工具，但是DevOps不建议将UI测试放在代码构建过程中调用执行，因为UI的变化元素比较多，很难及时维护一套稳定的UI测试代码，因此建议单独开发和维护UI相关的自动化测试系统。

除了在持续构建过程中的自动化测试之外，还需要根据具体项目的特点考虑何时需要加入性能测试和可用性测试，这部分需要单独开发专门的自动化测试程序，并单独安排相关的测试环境。

还有一部分测试可能无法用自动化测试来完成，只能用人工测试，主要包括系统演示性测试、探索性测试以及易用性测试等。

2. 自动化测试工具

根据不同的测试需求和运行环境，有很多自动化测试工具可供选择：QTP、LoadRunner、JUnit、JMeter、Selenium、Robort、AutoRunner等，如图3-15所示。

序号	工具	用途说明	备注
1	QTP	测试框架、功能测试、界面测试	商业软件
2	LoadRunner	性能测试	商业软件
3	Junit	测试框架、单元测试	开源软件
4	Jmeter	性能测试、功能测试	开源软件
5	Selenium	测试框架、功能测试	开源软件
6	Robort	测试框架、功能测试	开源软件

图3-15 自动化测试工具

Selenium是谷歌开源的自动化测试框架管理工具，可以做测试框架、测试案例管理、测试报告管理、测试场景定义等。Selenium是一种脚本语言，开发相对方便。Robort的内核也是Selenium，但做了一些故事场景定义的封装，从而更符合DevOps的自动化测试实

现。Robort 的测试管理过程如下：Robort 会提供两层的开发语言，在最上层提供一种描述性语言，需求人员会先在这一层里定义业务验收、测试的场景。这种描述性语言让需求人员可以很简单地定义每一步干什么，会做一个测试验收的故事场景。但其实这个开发并不能执行，因为此时只会自动生成一个代码框架。测试开发人员会在第二层里根据需求人员定义的测试开发框架去写具体测试实现的脚本，这就保证了自动化测试是按照用户故事场景的要求来定义的，是一个非常完美的验收测试方案。而且保证测试场景是按照最前端的业务和产品部门设想的要求去验收的。

目前，针对自动化测试存在一些误区：

- 只要使用自动化测试，就能缩短测试时间，提高测试效率。自动化测试的前期实现要花费更多的时间，相比创建和执行一个手工测试用例，要花费 3 ～ 10 倍的时间来开发、验证和文档化一个自动化测试用例。
- 自动化测试工具使用了图形化界面，很容易上手，对人员的要求不高。简单的录制 / 回放方法并不能实现有效的、长期的自动化测试，测试人员还需要对脚本进行优化，这就需要测试人员具有设计、开发、测试、调试和编写代码的能力，既有编程经验又有测试经验的候选人是最理想的。测试过程中还需要安排专业人员对测试脚本库中的脚本进行维护。

3. 持续测试小结

关于持续测试，我们总结了几点经验，供大家参考：

1）在 DevOps 流水线上测试团队是需要技能转型的一个团队。现在新增的测试人员我们希望是有开发背景的、能够做测试开发的人员。因为在原来手工代码测试的情况下，这条交付的流水线是“流”不起来的，那么敏捷的快速迭代和快速试错目标也就无法实现。所以，在整个 DevOps 的实施过程中测试团队的转型压力最大。

2）自动化测试是 DevOps 流水线上重要的一环。没有自动化测试也可以搭建流水线，但效率不会得到明显的提升。

3）自动化测试工具无法统一选择，需要根据测试的目的和程序技术环境来进行具体的分析。

4）自动化测试搭建起来之后，这套自动化测试的程序就应该进入软件版本管理范畴，这将是一个持续开发的过程。因此自动化测试程序开发的交付过程伴随在业务需求的开发交付过程中，将作为业务需求开发管理过程中的提交物的一部分进行统一的任务分配和管理。

5）自动化测试可以解决大量重复性测试工作的效率和质量问题。DevOps 认为工具就是用于提高效率、降低成本的。所以自动化测试本身作为工具解决的是效率和成本的问题，但它也能通过效率提高质量。一周时间人工可能只能完成一遍测试，但自动化测试可以完

成多遍，但最终是否能够真正提高质量，还要依据人工对测试场景、测试案例的设计。工具本身解决的是效率和成本的问题。DevOps 希望所有回归旧功能的测试验收过程都应该是自动化的，但探索性测试以及易用性测试可以做人工测试。

3.2.5 持续部署

1. 什么是持续部署

当有版本通过持续集成流水线进行构建之后，就可以将其部署至某个具体的环境，这就需要自动化部署技术，将这个自动化部署和持续集成流水线连接起来，就可实现持续部署。如图 3-16 所示，实现持续部署的前提是至少拥有一条完整的自动化构建、部署、测试和发布流程。

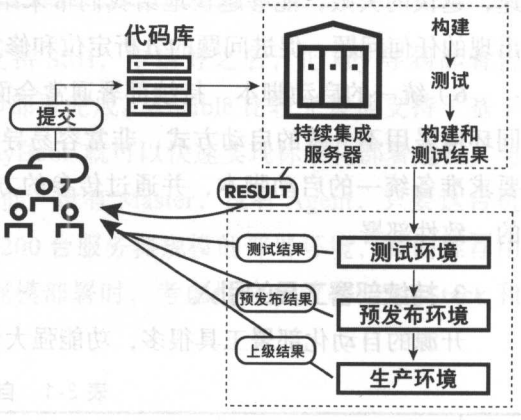


图 3-16 持续部署

传统软件的部署模式通常有如下几种。

- 通过纯手工的方式来部署应用软件。
- 在开发人员完成代码以后，才在生产环境做部署。
- 运维人员在生产环境通过手工方式修改配置。

这 3 种方式的执行效率较低，发布质量也没有稳定的保障，极大地限制了 IT 系统对业务的能力支撑，坚持传统部署模式，已经不能满足灵活多变的市场需求。DevOps 持续部署的适时出现，既解决了执行效率问题，又保障了交付质量。

2. 持续部署的关键要素

持续部署作为 DevOps 流水线上的重要环节，在实施的过程当中有哪些场景是需要重点关注的，我们来看一下。

1）部署的执行者必须参与创建部署的过程，也就是说工程师需要完全参与每一个具体的过程，亲身经历部署过程。

2）在整个部署过程当中，应该详细、明确地记录下每一个环节的所有经过以及相关活动的输入输出，通常不会选择去删除整个部署过程当中相关的或产生的任何数据，而是采用转移的方式，统一保存到规划好的地方，这样一方面方便做数据的回滚，另一方面也方便在对历史版本去做追溯的时候能够有据可查。另外，操作日志也需要写到日志卷上，根据规范不同，定义不同的日志级别和内容格式，然后设计定期的日志清理策略，以保证空

间的管理和数据的有效性。

3）持续部署是整个团队的共同责任，我们通常认为部署是运维人员的职责，而DevOps强调的是开发、运维一体化。每一次部署目标都是团队内的成员共同努力的结果，持续部署的本质是对团队交付成果的一次检验。

4）新的部署动作应该采用预发布的方式，在准生产环境或者测试环境预先部署并验证，避免给生产环境带来不可控的直接影响。对应的方法有蓝绿发布、金丝雀发布等模式。

5）积极面对快速失败。在整个持续部署的过程当中，我们鼓励去快速实施目标，快速执行部署动作，但快速频繁的部署很可能会带来快速失败的结果，那我们为什么要接受把快速失败当成持续部署的一部分呢？DevOps方法论里一直在强调敏捷，敏捷就代表着快速，越快的失败，能够越快地给我们带来结果的反馈，也方便我们能够尽快地发现过程中出现的任何问题，促进问题的分析定位和修复。

6）统一的启动脚本。持续部署通常会面临同一应用程序在不同阶段的不同环境，在不同环境采用不一致的启动方式，非常容易导致测试环境和生产环境的不一致，因此DevOps要求准备统一的启动脚本，并通过传参的方式来调用，从而保障同一应用程序在不同环境的一致性部署。

3. 持续部署工具的引入

开源的自动化部署工具很多，功能强大但又各有优劣，如表3-1所示。

表3-1 自动化部署工具

	Ansible	SaltStack	Puppet	Chef
简介	一个在远程节点上可重复性部署应用的开源配置管理工具	SaltStack是基于Python开发的一套C/S架构配置管理工具	Puppet是一种Linux、UNIX、Windows平台的集中配置管理系统，使用自有的Puppet描述语言，可管理配置文件、用户、cron任务、软件包、系统服务等	一个IT自动化工具，它把服务器的环境（软件、依赖库、网络等）进行抽象，以特有的配置语法（Ruby语言）对其进行管理，可以自动地进行服务器环境的初始化工作
优点	不需要安装特定的Agent，使用sshd 基于Python，易于学习社区非常活跃，增长迅速 简单清晰的PlayBook结构 可以通过JSON协议被各种其他应用调用	salt-ssh支持Agenttless模式C/S模式效率高YAML安装，易于学习优异的扩展性和弹性	资源抽象，不再关注软件依赖 社区成熟，有不少大企业用户 GUI功能完善支持的规模大 各种操作系统支持全面 安装配置简单 报表功能强大	模块丰富 资源抽象，不再关注软件依赖 社区活跃，目前采用chef的企业相对puppet少，但正在逐渐被接受 支持的规模大 可能通过JSON协议被各种其他应用调用 支 持Chef-solo和Server-Client两种管理方式

（续）

	Ansible	SaltStack	Puppet	Chef
缺点	功能不够强大，并必规模小，节点数超过500后效率逐渐降低 对Windows系统的支持还不成熟	官方WebUI功能较少 初始化阶段比较费时 对Windows系统的支持还不成熟	有自己的描述语言，使用复杂，有一定的学习成本，可移植性差，基于Ruby，相比基于Python的工具，自身性能稍差 Puppet的执行是无序执行，如果一些配置或部署有先后顺序，需要非常小心	基于Ruby的描述语言，有一定的学习成本，对于懂Ruby语言的人更容易 基于Ruby，相比基于Python的工具，自身性能稍差 不支持Push模式，而是client端定时从server端获取 安装配置较复杂，依赖特定中间件（如RabbitMQ、CouchDB）依赖于git

我们比较推荐的模式是：新机器只需要支持SSH，申请好之后，把key导到部署机器上，调用一个API主机申请，用sudo权限即部署完成。Ansible比较完善地支持了基于SSH的部署流程，而且插件丰富，通过编写PlayBook就可以快速实现标准化部署过程，与Puppet一类的部署系统进行比照，它的成本更低，没有Master，没有Agent，只要这台机器可以连上目标机就可以了，所以对于不超过200台服务器规模的部署环境，我们推荐用Ansible来实现持续部署和配置管理。但在大规模部署时，考虑性能和效率，SaltStack和Puppet则是更好的工具选择。

3.2.6 持续交付与容器化

容器化技术也同样可以应用到持续交付中，这会给持续交付带来什么样的变化？总体上我们可以用3个关键词来说明，分别是Build（构建）、Ship（部署）、Run（运行），这也是在整个持续交付过程中容器化所带来的好处的体现。最显著的好处是在打包过程中，如果在一个传统的非容器化项目里打包，包括代码、配置文件、环境变量配置等工作都需要人工设计、规划和操作；但如果以容器的镜像文件为对象来进行操作，只需要考虑注入参数，打包是更完整的、更彻底的，包括操作系统参数的打包，在打包的时候不仅进行了代码的编译，甚至完成了应用安装的过程，将应用程序和运行环境所需要的都进行了打包，最后整合成一个镜像文件。容器为持续交付带来了标准规范，非常有利于应用程序在不同环境间的流转和发布，提升了交付效率，这充分体现了DevOps里面所谓的配置即代码和基础设施即代码的核心理念。

如图3-17所示，容器化之后我们的部署对象不再是App或者是Java文件，而是容器镜像文件；而且也不再关注运行在什么操作系统下，操作系统如何配置。所以在搬运的时

候移动的是镜像文件，运行的时候是 Docker 镜像；只需要关注外面注入的环境参数即可，而且在运行过程中有诸如 Kubernetes 这样的管理工具来做统一的管理。所以，基于容器化的环境再做 DevOps，代码搬运、打包、运行监控等操作都已经把对象从 App 转到 Docker 容器上来了。这是一个从零配件组装到“集装箱式”搬运的变革，原来想要去定制一辆汽车的时候，需要考虑各种各样的零件如何组装，一个个地搬运零件，考虑车的参数，拿到所有的零件之后还需要一个详细的图纸，按照图纸安装并进行调试之后才能使用。现在是在工厂里就把车装好了，而且车需要的本地化涂装也一起放到集装箱里了，我们需要考虑的是如何去搬运整个集装箱。所以容器化带来的是整个持续集成效率的提升，这是一个组件化到整体化的质变。

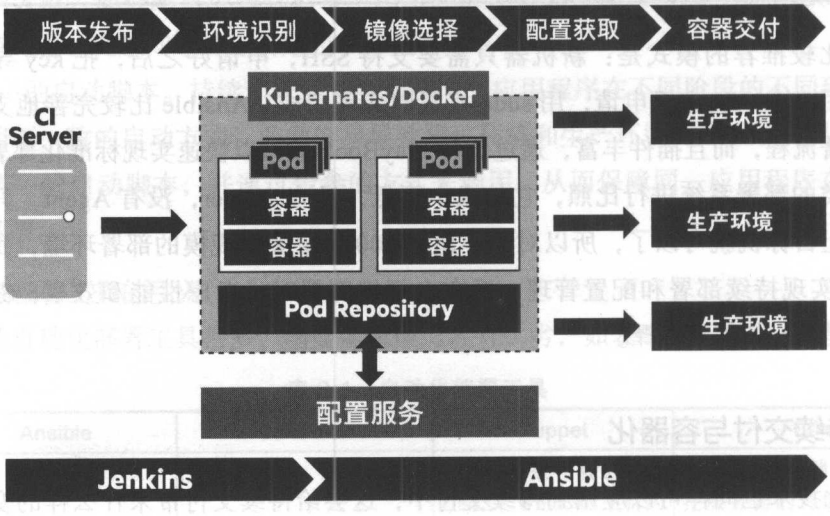


图 3-17　容器化持续交付过程

3.2.7　DevOps 实践框架总结

DevOps 的实践包含了软件生命周期的全流程，涉及了大量的开源工具，如图 3-18 所示，从需求到运维的每一个环节都有很多热门工具可供选择。典型的 DevOps 实践通常以 Jenkins 为核心，通过 Jenkins 来集成和调度 DevOps 流水线中的代码管理、构建、测试和部署。

当我们基于开源工具完成了 DevOps 流水线的构建，实现了流水线的持续运行和交付后，从管理层面还需要提供可视化的仪表盘，来帮助我们显性化流水线的状态和结果，透明化流水线的过程，持续优化并改进流水线，如图 3-19 所示。

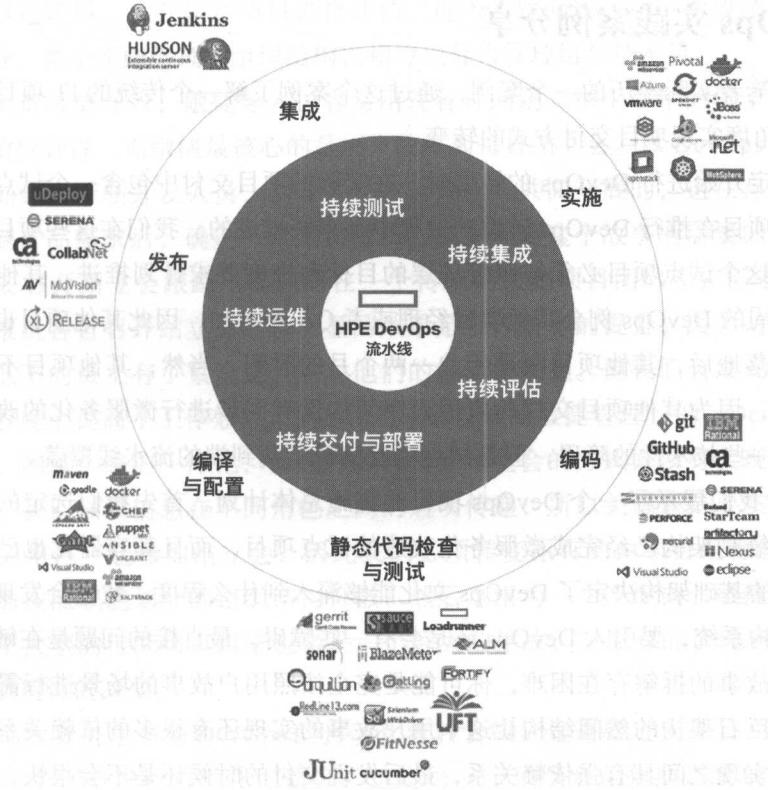


图 3-18 自动化流水线需要强大的持续集成工具的支撑

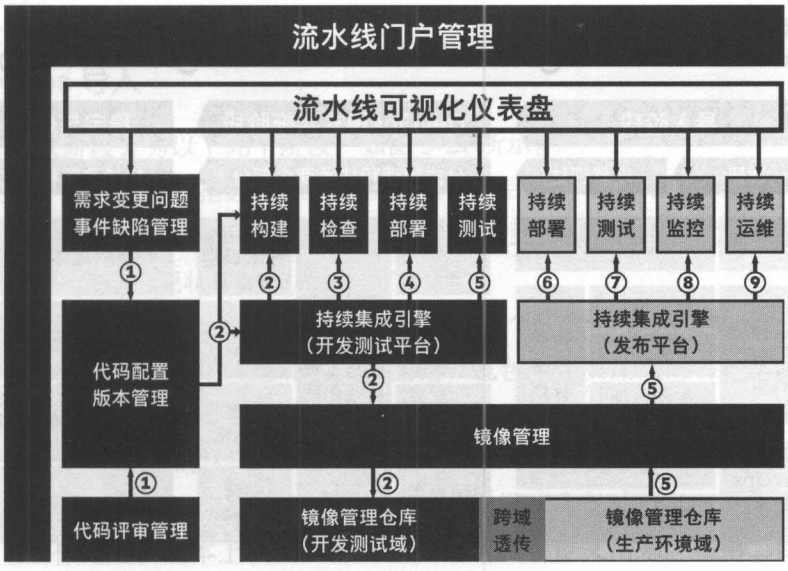


图 3-19 DevOps 流水线的建设蓝图

3.3 DevOps 实践案例分享

下面介绍笔者实际经历的一个案例，通过这个案例了解一个传统的 IT 项目或者组织如何从 DevOps 角度实现项目交付方式的转变。

当我们决定开始进行 DevOps 的转型时，在实际的项目交付中包含一个试点项目和若干其他项目，而项目在推行 DevOps 转型的过程中是多头推进的。我们在这些项目中指定了一个试点项目，这个试点项目必须按照管理层的目标和进度要求计划推进。其他项目也参与这个过程，每周的 DevOps 例会其他项目经理或者 QA 要参加，因此其他项目也是及时跟进的。试点项目落地后，其他项目跟进相差一两个月的时间。当然，其他项目不能保证流水线是端到端的，因为其他项目交付的应用程序架构没有同步进行微服务化的改造，因此在系统架构上有一些技术性的障碍，但是试点项目实现了端到端的流水线覆盖。

图 3-20 为我们展示了一个 DevOps 项目的实施总体计划。首先我们选定的试点项目所交付的软件系统是架构已经完成微服务化改造的试点项目，而且其容器化也已经基本完成了。整个项目的基础架构决定了 DevOps 文化能够深入到什么程度。我们会发现如果是一个传统的巨石架构系统，要引入 DevOps 还是会有一些障碍。最直接的问题是在敏捷交付迭代过程中，用户故事的拆解存在困难，你可能是完全按照用户故事的场景进行需求的分析和拆解的，但是巨石架构的然间结构让这个用户故事的实现还有很多的依赖关系，导致不同迭代或者功能实现之间具有强依赖关系，最后发现交付的时候还是不会很快，因为在代码的构建中打包的分支、合并依赖太多。

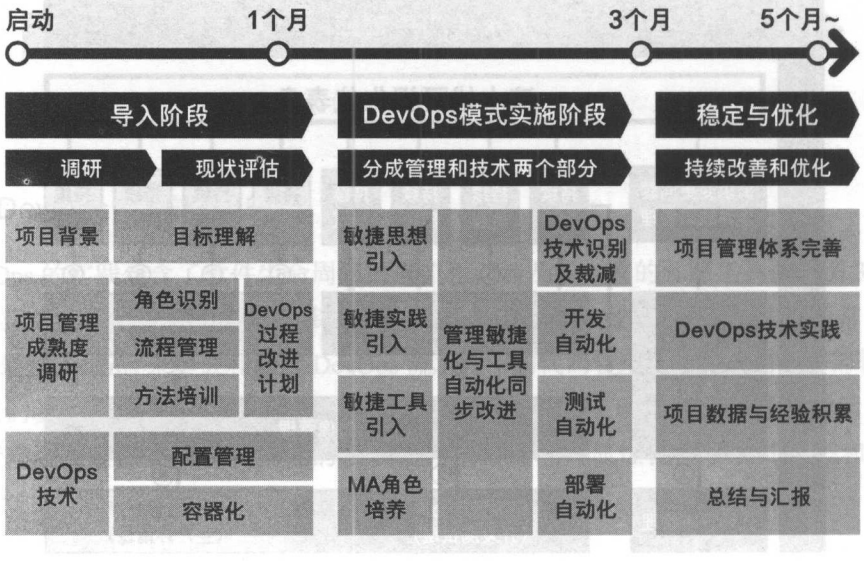


图 3-20 DevOps 项目的实施总体计划

试点项目选定后，就开始对项目进行评估，进入实践阶段，这一阶段基本上分成管理和技术两部分，整个实施管理的组织结构也相应地分为管理组和技术组。

管理组负责敏捷导入，敏捷导入也容易出现各种问题。举个简单的例子，一个大型的项目组引入敏捷管理，而敏捷最核心的是用户故事和每日站立会。以 DevOps 的思想典型的站立会是产品经理带领开发人员、测试人员、运维人员共同开展的，这在互联网项目里不是问题，收到产品需求后，确定一个用户故事，成立实现这个故事所需要的敏捷团队，然后每日就需要召开站立会跟踪工作。而在一个传统的大型项目团队中，需求组、开发组、测试组、运维组各自召开站立会，最明显的效果是工作效率的提升，因为站立会意味着透明化，效率低下的员工有了紧迫感，这对他们的触动很明显。但我们会发现这样的站立会虽然在一定程度上提高了工作效率，但达不到 DevOps 的敏捷管理目的，DevOps 的目的是需求、开发、测试、维护一体化，而这种各自召开站立会的方式并没有实现各个环节的统一部署调配。我们强调信息在不同角色之间的透明传递，所以关键还是要实现所有参与这个用户故事的角色一起参加站立会，因此我们必须转变原有的由项目组管理的组织结构，将原来巨大的按照职能划分的组织结构打散，然后重组为一个个纵向、敏捷的小团队来工作。这对一个大型的传统项目组来说会是一个巨大的挑战，对原有的管理制度、流程和分工都是一个颠覆性的改变。

技术组则负责搭建一条 DevOps 的自动化流转管线，这需要定义项目交付的过程，在此基础上对每个环节进行自动化的研究，选定每一个环节自动化所需要的工具，再利用 Jenkins 这样的工具将所有的工具集成并衔接起来，让它们成为一条一体化、单件流的 DevOps 流水线。

3.3.1 DevOps 导入

DevOps 导入阶段分为以下几个阶段，如图 3-21 所示。

1）阶段一：建立基线，搭建团队，明确责任。然后进行现状评估和差距评估，通过调研的方式将现状和目标表格做出来。在这个阶段需要用到 DevOps 成熟度模型表格，用这个成熟度模型的定义来进行现状评估和目标设定。

2）阶段二：思想提升，实施培训。培训分为多个部分，首先是敏捷导入，其次是 DevOps 培训，既包括 DevOps 理念和实践方法论，也包括需要用到的各种工具的安装、配置和使用方法等。通过培训既要让大家对敏捷、对 DevOps 进行整体了解，也要让大家对 DevOps 的推进路径有清楚的认识。

3）阶段三：细化计划。定义整个流水线的具体目标，确定时间表。

完成以上工作之后，基本上我们具体要做的事情就会定义出来。在第二阶段具体实践的时候就可以围绕总体计划来开展工作。

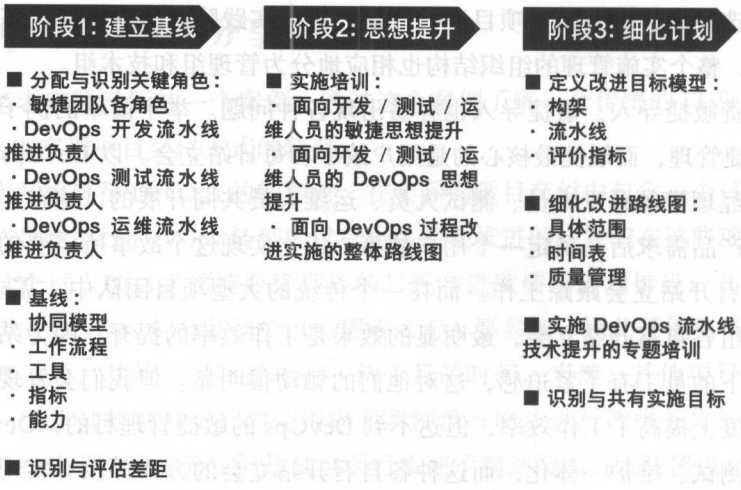


图 3-21 DevOps 改进实施方案——改进计划（导入阶段）

3.3.2 DevOps 实施

承接导入阶段，DevOps 实施阶段分为以下几个步骤进行，如图 3-22 所示。

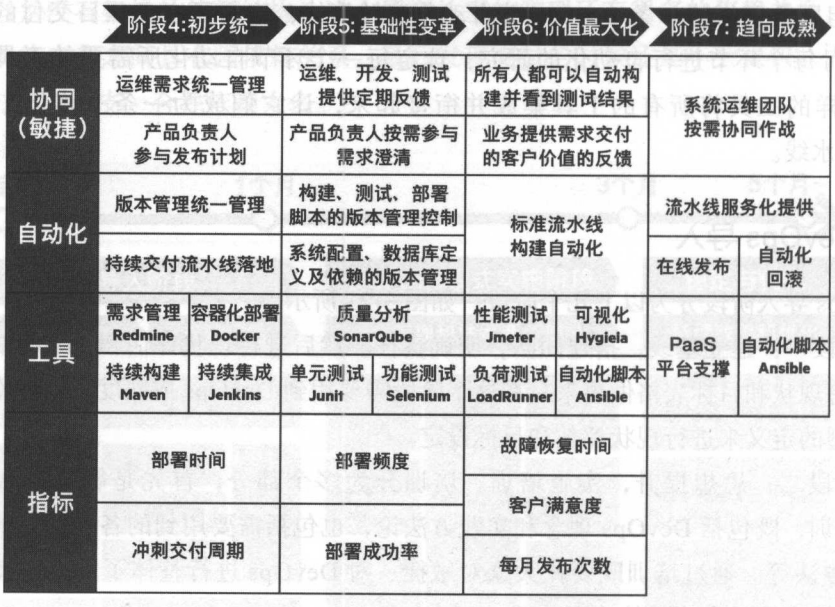


图 3-22 DevOps 改进实施方案——改进计划（实施阶段）

1）阶段四：初步统一。首先需要将需求的版本管理起来，这个版本是在需求管理的基础之上进行的，包括需求的录入、用户故事的拆解、需求的分配和领取，敏捷团队的任务

创建和跟踪以及敏捷团队的白板、燃尽图管理等内容。然后开始搭建持续集成流水线。这个阶段引入了几个工具，如需求管理工具 Redmine、容器化部署工具 Docker、持续构建工具 Maven、持续集成流水线工具 Jenkins。并且在这一阶段将代码的版本管理搬到流水线上去，将代码的构建及时引入自动化测试。

2）阶段五：基础性变革。SonarQube 的引入对于整个代码的透明化管理很明显。在这个阶段开始引入单元测试和功能测试，在作者所经历的这个实践项目的初期，我们用 JUnit 做单元测试，用 Selenium 做功能测试。

3）阶段六：价值最大化。持续进行自动化测试的开发，自动化测试是一个从无到有的过程，案例要逐渐去覆盖。我们同时在这一阶段引入 Hygieia 可视化仪表盘。可以通过 Hygiea 看到最核心的工具所产生的报告，如 Jenkins、SonarQube 等生成的报告，因此能看到流水线上各个团队的效率和效果，但 Hygieia 本身也存在一些问题，如不能将 Ansible、Redmine 集成进去。但 Hygieia 是开源的，我们可以对 Hygieia 进行改造，使得需求报告集成进去。这一阶段还进行了自动化部署，Ansible 在不同项目里有不同的应用，在容器化项目中，Docker 镜像包装的过程用 Ansible 自动打包。

4）阶段七：趋向成熟。系统运维团队按需协同作战。

在开发的过程中，我们引入敏捷站立会。按照站立会的要求每天召开，最初可以是物理看板，如白板或是标签，在大家对每天站立会已经接受并习惯了之后，可以迁移到电子看板上，如图 3-23 所示。

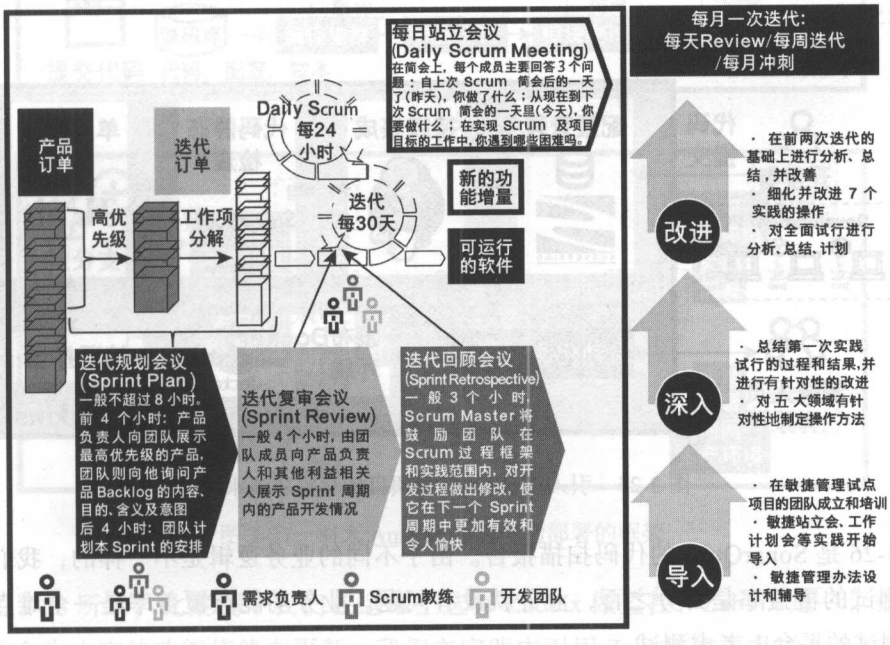


图 3-23 开发过程引入敏捷 Scrum

在项目管理中引入 Redmine 进行管理，基本上所有的项目管理都可以在其中进行，如图 3-24 所示，如需求录入、生产问题录入、测试问题录入、生产版本定义、需求或者问题任务分配、子任务分解、任务状态跟踪等。但我们在用 Redmine 的时候要关注到底选择什么样的项目管理文化，这决定了工具里的流程配置方案。

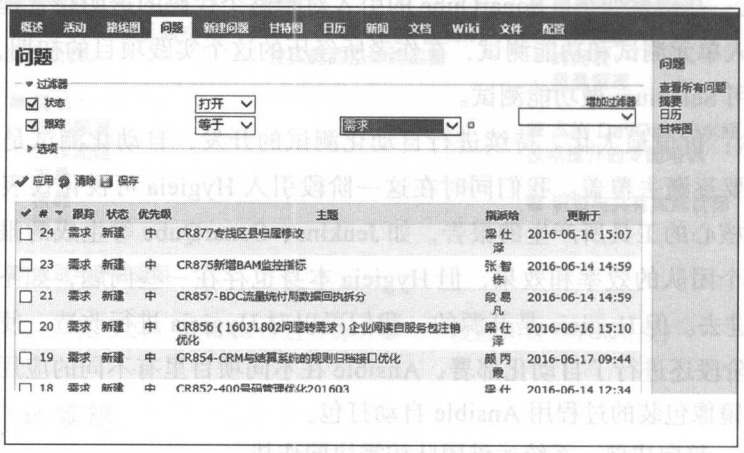


图 3-24 在项目管理中引入 Redmine 进行管理

用 Jenkins 搭建流水线，项目通过 Jenkins 来搭建持续集成平台，实现代码自动化编译、单元测试自动化、代码扫描工具的集成、自动打包的发布、自动化功能测试平台的打通等，如图 3-25 所示。

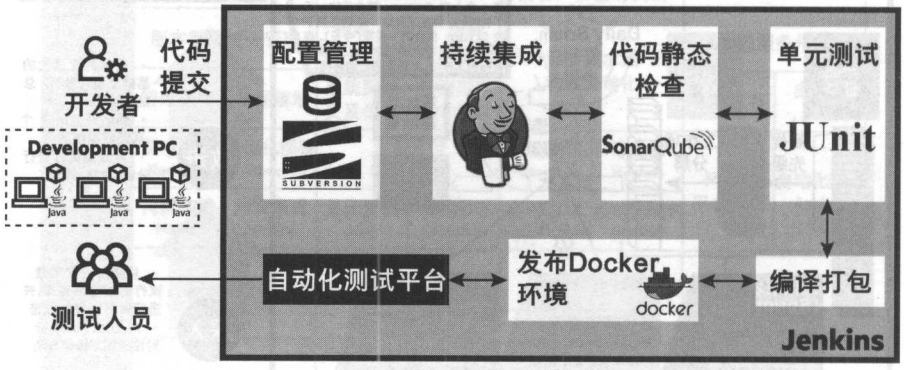


图 3-25 引入 Jenkins 搭建项目持续集成流水线

图 3-26 是 SonarQube 的代码扫描报告。由于不同的业务逻辑是不一样的，我们无法确定单元测试的覆盖率是百分之百，业务就没有问题。业务功能点覆盖率是一个难点，需要在功能测试的平台上考虑测试。

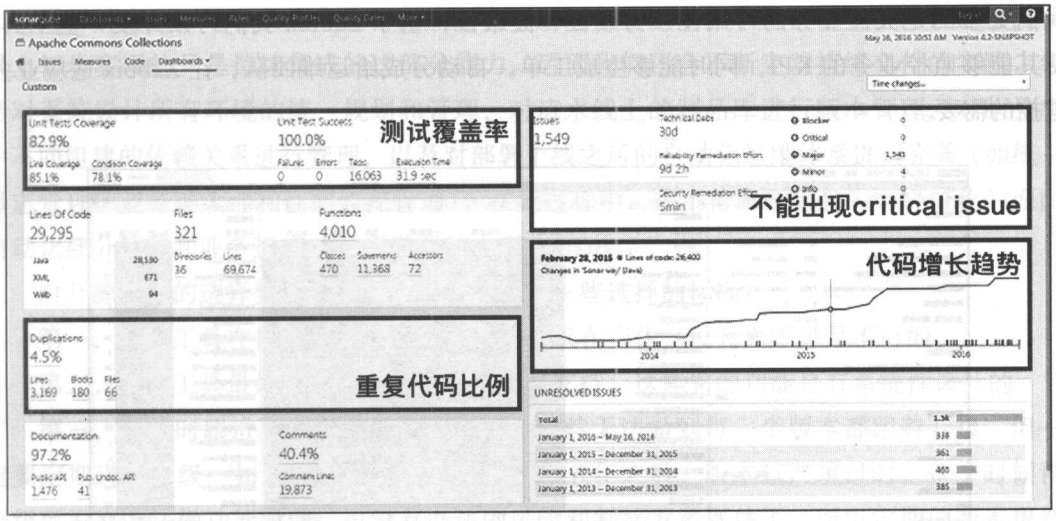


图 3-26 引入 SonarQube 作为静态代码扫描工具

图 3-27 是 Ansible 部署的结构框架，Ansible 用来做 Docker 的打包。自动化部署有一个可以界面交互配置管理的工具，而且开源工具之间是无缝集成的。

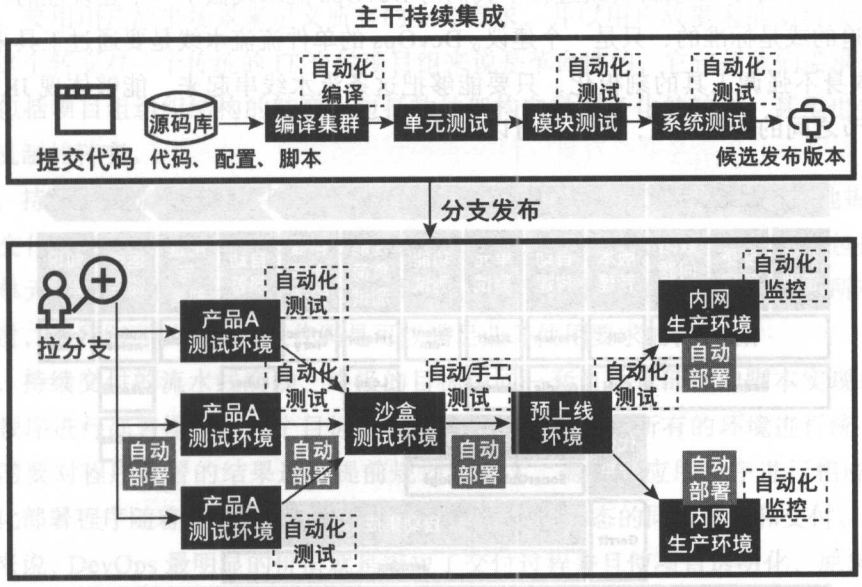


图 3-27 引入 Ansible 作为自动部署的框架

如图 3-28 所示，此案例中我们在运维上选择 Zabbix 做监控，Zabbix 是一个很老的工具，但迄今为止它的功能依然在更新。我们在实践中引用 Zabbix 主要是做业务层面监控，

我们更多的是关注业务的可用性、持续性、健康性，基于 Zabbix 我们可以开发一些工具，使其能够监控业务的 KPI，同时能够生成工单，自动分成给运维团队，让 Zabbix 适应业务监控的需要。

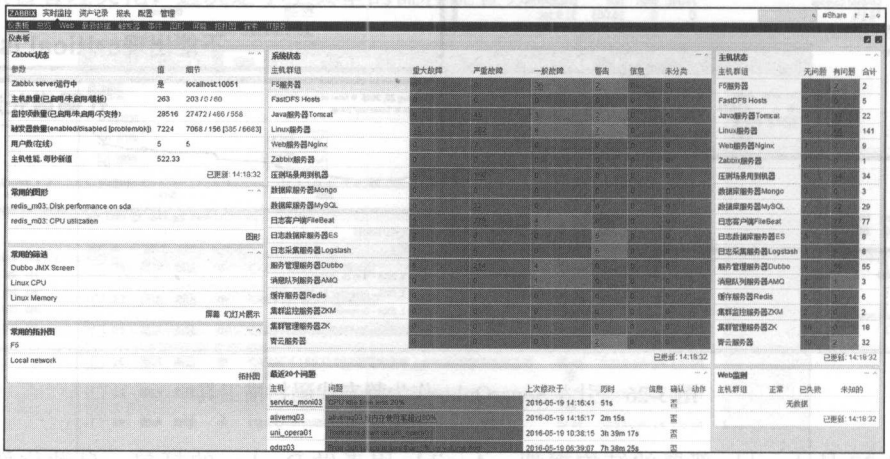


图 3-28　引入 Zabbix 搭建自动化监控平台

如图 3-29 所示，在项目实践过程中我们对 DevOps 流水线做了一个建议性的环境定义，这不是强迫的或是标准的，只是一个建议。DevOps 的单件流流水线是要通过工具来实现的，DevOps 本身不强调工具的刻板化，只要能够把这条流水线串起来，能够体现 JKK 思想的流水线环节之间的衔接定义，那么都可以选择。

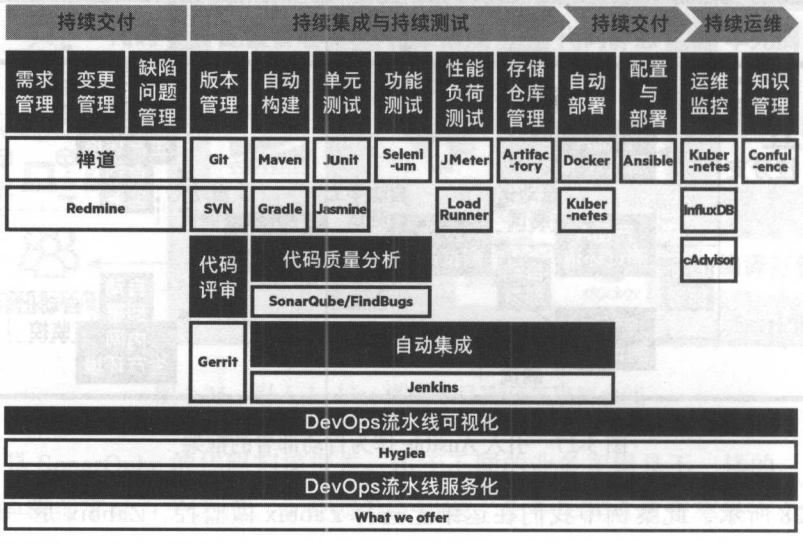


图 3-29　DevOps 流水线实践总结

到目前为止，笔者的这个项目仍然还在不断地完善这条流水线，事实上，在这条流水线上初步实现的只是持续集成的这部分能力，后面我们还会不断完善持续交付的实践，包括对系统设计所有环境的统一规划和管理，对流水线上的制品库进行版本管理，对制品库中不同组建的依赖关系进行管理，以及对部署上线之后的自动化运维体系进行完善（如统一日志管理、业务连续性和性能监控管理）。在此过程中，还不得不建立起一个轻量化的面向自动化部分管理和业务运行监控所需要的小型 CMDB。

对开源工具的选择要持开放的心态，但也有一些选择的标准：

第一，这个工具在社区里是热门的，因为需求在变化，没有人跟进是不行的。

第二，开源工具最好有一些大的厂商在背后支持，这样它的功能是有系统性设计的。

第三，要及时跟进开源工具的版本更新，虽然无法做到每一个版本都跟着升级，但一定要定期进行升级。如果没有及时进行升级，三五年之后，版本就严重过时，这个时候再升级就没有办法做功能对照，也没有办法做风险和割接方案评估了，所以定期的版本更新要跟上。

以上 3 点是我们在开源社区选择工具的基本原则。

就笔者自己的实践来说，虽然我们的 DevOps 转型仍然还在持续进行中，但是我们的转型有几个简单清晰的终极目标。

首先，要用用户故事场景来定义所有的业务需求，并以用户故事来推动后续的迭代交付过程，这个转变对一个传统的 IT 交付项目组来说是革命性的，它包括我们需求分析方法的转变，包括项目组组织结构的转变，包括软件架构向微服务化的转变，甚至也包括传统项目考评机制的转变。

其次，持续集成流水线的实践、终极的评价目标让开发团队可以更加频繁地提交代码，而每次提交代码之后，可以在 5 分钟之内得到一次集成的结果报告，这个报告包括了打包的结果、单元测试的结果、代码扫描的结果，也包括了验收测试的结果。如果所有的报告都显示通过，那我们可以认为这段代码是可以满足业务使用要求的代码。

最后，持续交付的流水线实践、终极的目标用同一套自动化部署的脚本实现对所有环境的应用程序进行部署操作。这个目标的实现需要我们提前对所有的环境进行统一的规划和定义，需要对程序部署的结果进行提前规划和定义，需要对应用程序进行相应的改造，需要自动化部署程序随着业务需求的开发和交付一起进行动态的同步开发和交付。

总体来说，DevOps 最明显的价值就是缩短了交付过程并且使项目透明化，原来在开发、测试过程中这个环境可能是黑盒，但 DevOps 使得我们能看到整个项目的进展，能看到效率的提升。另外，DevOps 还保证了代码的质量，降低了风险。

Chapter 4 第 4 章

Docker 快速入门

当前业界精通容器技术的高级架构师非常奇缺，一方面普通的架构师不是很懂容器技术；另一方面在很多人眼里 Docker 是属于运维的，认为运维工程师懂就够了，而运维工程师又不懂架构，因此造成当前这种人才短缺的局面。

Docker 自从开源以后就迅猛发展，早已超越其最初的目标——规范化和自动化软件打包和发布流程。当前 Docker 正在向两个领域延伸：一个是替代虚拟机技术，内嵌容器引擎的新一代操作系统正在发展成为新的 IaaS 基础设施；另外一个是以容器技术为基础的新一代分布式架构与微服务平台技术，它们蓬勃发展，最典型的是谷歌开源的 Kubernetes 微服务架构，这是一个类似当年的 J2EE 并且更为强大的新平台。2015 年年底中国移动部署了国内最大的 Kubernetes 线上生产环境，相信越来越多的传统系统会 Docker 化，并且进一步实现微服务化改造。此外，即使我们不在生产环境中使用 Docker，Docker 也对我们学习新技术、可持续集成、架构选型测试、项目 POC 等具有切实的价值。

本章的内容大致分为四部分，由浅入深，目的是实现 Docker 快速入门的学习目标。首先是 Docker 概述，通过回顾容器技术的兴起过程，讲述 Docker 的价值，介绍幕后的主导力量以及蓬勃发展的生态圈，得到一个再明显不过的道理，即我们需要学习并掌握 Docker 这门技术；其次，了解 Docker 的几个关键概念及术语，并实践 Docker 几个基本的命令行操作，以便熟练应用 Docker；第三，我们通过一个例子来学习如何用 Docker 技术改造传统项目；最后一部分是 Docker 高级进阶的内容，讲述 Docker 背后的原理技术以及 API 编程接口。

4.1 Docker的价值及生态圈

我们学习一门新技术，以及决定在公司推广并使用这门新技术，要付出学习成本并且要承担一定的风险，由于IT行业中很多新技术都是昙花一现，很快就成了明日黄花，很难带来相应的回报。因此，除了深刻认识新技术本身的价值之外，我们还必须了解这门新技术背后的生态圈和推动力，确保我们所要花费大量时间和精力去学习的新技术是有旺盛生命力的。本节我们就从Docker的价值、Docker的背景和生态圈等方面去全面了解它。

4.1.1 Docker的价值

如何正确理解Docker这个新技术的价值呢？下面我们以软件开发活动中常见的一个任务来解释和发掘Docker的价值：

“在某个Linux服务器上部署一个MySQL Server 5.7实例并创建一个库，供项目开发测试使用。”

对于MySQL DBA或者熟练的运维工程师来说，这是一个非常普通的任务，但是对于其他人来说可能是一项颇具挑战的工作。在实现上述任务的过程中，可能遇到各种意外：例如，服务器上已经有一个低版本的MySQL了，此次新安装一个5.7的高版本实例时就比较麻烦了，这里涉及一个所谓隔离的问题，因为Linux下有很多开源软件不具备隔离性，一般不能同时安装几个实例。比较直接的解决方式是，从源码编译实现定制安装，但这个过程对技能的要求就更高了。此外，Linux下的二级制安装包经常要依赖大量的其他基础包，很多时候还要求严格版本匹配，否则就无法安装成功。这其实是Linux的一个痛点，即使YUM仓库体系也没能完全解决这个问题，与这个问题相关的另外一个痛点是同一个应用程序在不同的Linux发行版上的安装部署无法做到统一。所以，即使是安装很常见的MySQL数据库这样的基础软件，是否能安装成功，需要多长时间，很大程度上取决于系统环境以及任务执行者的经验和能力，你无法准确预估完成此任务究竟需要半小时、半天或者是3天。那么，如果用Docker来完成此任务，究竟需要多少时间呢？答案是5分钟就好，只要一个命令行：

```
docker run -d --name mysqlsrv1 -p 3306:3306 -e MYSQL_ROOT_PASSWORD=123456  mysql
```

上述命令的意思是启动一个名字为mysqlsrv1的MySQL容器，MySQL的ROOT用户密码为“123456”，并且绑定了主机上的3306端口，因此我们可以用安装Docker主机的IP地址以及3306端口访问这个容器化的MySQL Server，用起来与普通的MySQL Sever完全一样。然后我们还能用docker exec命令进入到容器内部进行操作，如查看MySQL的配置信息、数据文件，以及相关进程状态等信息。如果在一个主机上部署多个版本或者相同版本

的 MySQL，只要绑定不同的主机端口即可让它们和平共处并共同提供服务了。

从上面的过程可以看出 Docker 容器具备“即插即用”的特点——直接启动就可以用，而没有我们所熟悉的传统的复杂“安装过程”。这是为什么呢？答案就在 Docker 容器所使用的 Docker 镜像（Docker Image）里。

因为 Docker 镜像是把安装结果提前“固化”的一种特殊二级制程序包，这种特殊的镜像文件通过 Docker Build 工具自动执行安装脚本，并且把安装结果“固化”到一个特殊的压缩包里，这个压缩包内部包括了主程序运行过程中所需要的所有依赖程序与配置文件。Docker 镜像与 Linux RPM 包不一样，因为 RPM 不包括它所依赖的 RPM 包，而 Docker 镜像完全是自包含的，不需要任何第三方镜像，而且任何两个镜像之间没有依赖性，运行期不依赖于环境。因此，Docker 镜像做到了与外部环境无关的隔离性，在任何一个安装了 Docker 运行环境的 Linux 系统中都能 100% 正确运行，也就是说任何人只要按照上面的 Docker 指令操作，都会是完全一样的结果，不再依赖具体任务执行者的经验和水平，所以 Docker 技术其实把一些运维工作标准化了。因此在 Docker 的帮助下，一个新人可以很容易地达到一个具有 3 年经验的运维工程师的水平，也具备了在短时间内快速掌握更多常用中间件和基础设施的安装部署工作的能力。此外，由于 Docker 所具备的环境无关性，以及资源占用少的优势，我们可以在自己的笔记本式计算机中启动一个虚拟机来学习和检验 Docker 化的程序部署过程，只要在本地的虚拟机中运行成功了，那么在任何服务器中都会成功。

如果从整个软件项目的开发流程来考虑，第一步应该是环境搭建，包括开发环境、集成测试环境、用户验收环境、线上生产环境等，每当有新版本发布之后，部署并应用到这些环境中又是一个占用很多时间的重复性体力劳动，由于大量工作都是手工来完成的，因此这种任务的完成时间和质量是不确定的，是取决于具体的人与环境的。互联网领域里有很多常见的中间件，如 Redis、MemCache、RabbitMQ、Kafka、ZooKeeper 等主流的中间件已经超过 20 个，基本上现在搭建一个稍微大点的分布式互联网软件平台，可能就会用到 5 个以上的开源中间件，这些中间件相对于 MySQL 来说，技术上都比较新，通常还是集群方式，因此仅仅成功部署这些中间件，就是一个极具挑战的任务了。Docker 的价值在这里不言而喻，这也解释了为什么 Docker 一开始是在互联网领域兴起的，因为 Docker 公司早已把这些常用的中间件做成镜像了，因此只需要几个简单的 Docker 命令，运维人员就可以在短时间内快速完成中间件的部署，成倍地提升效率，所以 Docker 一开始就吸引了大量互联网公司的注意。

Docker 技术不仅促进了运维标准化，而且也实质性地促进了开发流程以及 PaSS 平台的自动化。OpenStack 也在服务编排自动化方面做了很多努力，与此同时，其他一些大的 PaaS 平台也希望实现应用部署、扩容、升级等任务的自动化能力。但无论 PPT 或演示效果

多么震撼，却始终难以真正落地，这是因为生产环境中所用的独立软件组件的安装部署是非常复杂的，很大程度上依赖于具体的环境，不同的软件组件版本还会引起部署脚本的细微差别。虽然理论上可以编写一个复杂的安装脚本来适应任何环境与版本的组合，但实际上这个脚本很难编写并被持续维护，这也就是为什么在 Docker 之前的软件自动化部署无法真正用于生产的最主要原因。Docker 通过制造镜像的方式规避了安装这一最复杂的问题，所以 Docker 化的软件就不存在安装环节了，因此可以很方便地在任意机器上启动和重启。此外，借助于 Docker Engine 所提供的 REST API，诸如镜像打包、镜像上传、镜像下载、容器创建、容器全生命周期管理等任务都能通过编程方式来实现自动化驱动，从而实现了软件全生命周期过程中的自动化驱动，进一步提升了敏捷开发的能力。

总体来说，Docker 使我们可以快速方便地安装部署复杂软件。对整个团队而言，可以节省成本并提升工作效率；对个人来说，掌握 Docker 有助于我们快速部署云计算和大数据领域里的众多新的中间件，从而快速学习并掌握它们的用法，深入 API 开发学习，节省时间成本。

4.1.2 学习 Docker 需要多长时间

如果要学习 Docker，那么需要多少时间呢？

答案是：只要一天。

首先，Docker 的安装非常简单，Docker 本身依赖的内核周边组件非常少，所以它的安装几乎没有任何问题，建议使用 CentOS/RedHat 系列的操作系统，最新版的都集成了 Docker YUM 源，因此只要一个命令 yum install docker 即可安装 Docker 环境。

安装好 Docker 以后，就可以通过下面的命令查找有哪些镜像可用，如查找 Tomcat 的镜像：

```
[root@hpeu-docker-k8s ~]# docker search Tomcat
NAME                          DESCRIPTION                                     STARS
OFFICIAL   AUTOMATED
Tomcat                        Apache Tomcat is an open source implementa...   1061
[OK]
dordoka/Tomcat                Ubuntu 14.04, Oracle JDK 8 and Tomcat 8 ba...   27
[OK]
consol/Tomcat-7.0             Tomcat 7.0.57, 8080, "admin/admin"              16
[OK]
consol/Tomcat-8.0             Tomcat 8.0.15, 8080, "admin/admin"              15
[OK]
cloudesire/Tomcat             Tomcat server, 6/7/8                            11
[OK]
davidcaste/alpine-Tomcat      Apache Tomcat 7/8 using Oracle Java 7/8 wi...   10
[OK]
```

```
    andreptb/Tomcat                Debian Jessie based image with Apache Tomc...   6
[OK]
    openweb/oracle-Tomcat          A fork off of Official Tomcat image with O...   4
[OK]
    fbrx/Tomcat                    Minimal Tomcat image based on Alpine Linux      3
[OK]
    kieker/Tomcat                                                                  2
[OK]
    picoded/Tomcat                 Tomcat 8 with java 8, and MANAGER_USER / M...   1
[OK]
    camptocamp/Tomcat-logback      Docker image for Tomcat with logback integ...   1
[OK]
    abzcoding/Tomcat-redis         a Tomcat container with redis as session m...   1
[OK]
    bitnami/Tomcat                 Bitnami Tomcat Docker Image                     1
[OK]
    stakater/Tomcat                Tomcat based on Ubuntu 14.04 and Oracle Java    0
[OK]
    inspectit/Tomcat               Tomcat with inspectIT                           0
[OK]
```

然后就可以使用 docker run 命令启动一个 Tomcat 实例了：

```
[root@hpeu-docker-k8s ~ ]# docker run -d -p 8080:8080 Tomcat
b3a839b2e739d8661fb53772f275306391bfd868941de576df3f1ecafd6de63a
```

假如虚拟机的 IP 地址是 192.168.18.3，则可以在笔记本式计算机上打开浏览器，访问 http://192.168.18.3:8080 端口，就能看到 Tomcat 页面了，如图 4-1 所示。

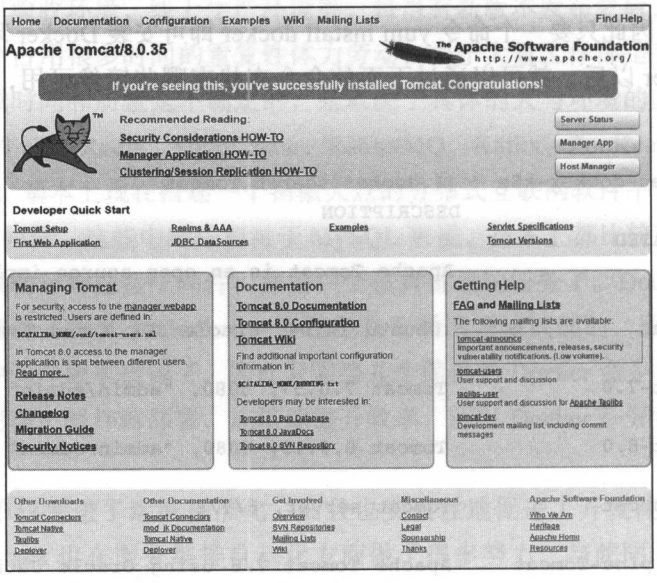


图 4-1 Tomcat 页面

接下来，可以用下面的命令查看我们启动的 Docker 容器列表：

```
[root@hpeu-docker-k8s ~ ]# docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED
STATUS              PORTS                    NAMES
b3a839b2e739        Tomcat                   "catalina.sh run"        7 minutes ago
Up 7 minutes        0.0.0.0:8080->8080/tcp   awesome_visvesvaraya
```

容器 ID（Container ID）是容器的唯一标识，可以用下面的命令进入 Tomcat 容器中排查问题，以及了解容器的运行情况，如查看容器的 IP 地址以及容器内运行的进程：

```
docker exec -it b3a839b2e739 bash
root@b3a839b2e739:/usr/local/Tomcat# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
4: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether 02:42:ac:11:01:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.1.2/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:102/64 scope link
       valid_lft forever preferred_lft forever
root@b3a839b2e739:/usr/local/Tomcat# ps -efwww
UID         PID   PPID  C STIME TTY          TIME CMD
root          1      0  2 00:40 ?        00:00:19 /usr/lib/jvm/java-7-openjdk-
amd64/jre/bin/java -Djava.util.logging.config.file=/usr/local/Tomcat/conf/logging.
properties -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
-Djdk.tls.ephemeralDHKeySize=2048 -Djava.endorsed.dirs=/usr/local/Tomcat/endorsed
-classpath /usr/local/Tomcat/bin/bootstrap.jar:/usr/local/Tomcat/bin/Tomcat-juli.
jar -Dcatalina.base=/usr/local/Tomcat -Dcatalina.home=/usr/local/Tomcat -Djava.
io.tmpdir=/usr/local/Tomcat/temp org.apache.catalina.startup.Bootstrap start
root         44      0  0 00:52 ?        00:00:00 bash
root         52     44  0 00:53 ?        00:00:00 ps -efwww
```

我们看到容器的 IP 地址是不同于 Docker 主机上的虚拟二层网络 IP 地址的，容器内也只运行了一个 Tomcat 的主进程，同时在容器内也看不到主机上的其他进程，这就是容器隔离机制的一个表现。

容器如果暂时不用了，可以挂起（docker pause <containerId>）或者停止（docker stop <containerId>），挂起或者停止的容器可以恢复运行，而容器中的数据仍然存在，如果是永久性地不使用了，则可以删除容器（docker rm <containerId>），此时容器所占用的磁盘空间也会被清理。

有时候我们需要把一个容器持久化成镜像，即通过启动一个容器，并且在容器中执行

一些 Linux 命令来完成某个软件的安装配置，完成以后，将其“固化”为一个新的镜像，这时我们可以用下面的命令实现：

```
docker commit <containerId>  newImageName
```

然后，这个新镜像则可以通过 docker save 命令导出为本地压缩包：

```
docker save <imageName >   <imageName.tar>
```

把镜像压缩包复制到另一台机器上，就可以使用下面的命令加载到 Docker 本地镜像中了：

```
docker load <   <image.tar>
```

然后用 docker images 命令可以列出本地所有的镜像：

```
docker images
REPOSITORY                                  TAG                 IMAGE ID
CREATED             SIZE
myweb_image                                 v2                  be08f59582d6
11 weeks ago        358.2 MB
myweb_image                                 v1                  7bff4a006636
11 weeks ago        358.2 MB
centos                                      latest              904d6c400333
5 months ago        196.7 MB
Tomcat                                      latest              51e1432889af
5 months ago        357.2 MB
mysql                                       latest              2fd136002c22
5 months ago        378.4 MB
```

不再使用的镜像，可以删除（docker rmi <imageName>），如果一个镜像被某个容器使用就无法删除，即使这个容器已经停止运行，我们需要先删除对应的容器，用 docker ps -a 可以列出所有的容器，命令结果中会同时给出容器的状态，如运行中或者是已经结束：

```
docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS               NAMES
b3a839b2e739        Tomcat              "catalina.sh run"        52 minutes ago
Exited (143) 4 seconds ago                      awesome_visvesvaraya
```

只要掌握了上述几个基本的 Docker 命令，Docker 就入门了，后面可以边学边用，只要几周时间就能达到很熟练的程度了。对有些人来说，学习使用 Docker 还是有难度的，这是因为 Docker 技术是建立在 Linux 上的，必须要掌握基本的 Linux 命令才能玩转 Docker，考虑云计算里的重要技术全部都是 Linux 领域内的，所以学习并熟练掌握 Linux 开始变得重要起来。

4.1.3 Docker是什么

Docker虽然以Linux容器技术为底层核心，但Docker本质上其实是一个PaaS平台，回到Docker诞生的年代可以更清晰地看清这个问题：2010年，几个年轻人在旧金山成立了一家做PaaS平台的公司——dotCloud，也用到了Linux容器，与其他PaaS平台不同，dotCloud把需要花费大量时间的重复性工作（如中间件和基础设施的安装）抽象成不同组件和服务，并且规范了Linux容器的格式，即后来的Docker镜像，提供了镜像库的全球仓库——Docker Hub，从而让开发者可以打包他们自己的应用以及依赖包到一个可移植的镜像中，并且提交到Docker Hub上，然后发布到全球任何一台Linux机器上启动并运行。无论是本机电脑还是公有云中的虚拟机，dotCloud还提供了便利的容器生命周期管理工具和API接口，方便开发者管理监控自己的应用。虽然dotCloud公司的这个产品很不错，但PaaS平台从来没有成为一个“靠谱”的软件市场，没有哪个公司的PaaS平台卖得非常好，因为每个稍大规模的软件公司都觉得自己能做PaaS平台，后来dotCloud公司只得将其PaaS产品开源了，开源以后就称之为Docker，而dotCloud公司也改名为Docker，并成为这几年最风光的开源公司。

如图4-2所示，Docker巧妙地解决了传统PaaS平台以及软件开发过程中的一个痛点——在构建基础框架，准备测试环境、版本部署等过程中所产生的大量人工重复劳动的问题。在整个软件开发过程中，准备测试环境和版本部署贯彻始终，由于严重依赖于环境和人，几乎在所有的大型项目中都会带来诸多问题，并且系统规模越大，带来的影响越大，所以它们成为项目中最不可靠的因素。

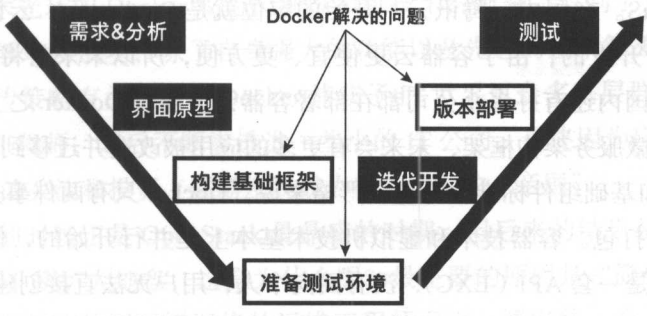


图4-2 Docker是什么

Docker作为一个先进的PaaS平台，它的几个关键特性如图4-3所示。

1）开放性。Docker本身是开源并面向整个IT行业开放的，IT人员都可以参与，Docker Hub里80%以上的镜像是由全球各地的运维工程师提供的，基本上我们能想到的任意一个开源中间件和基础设施都能在Docker Hub上找到对应的Docker镜像。

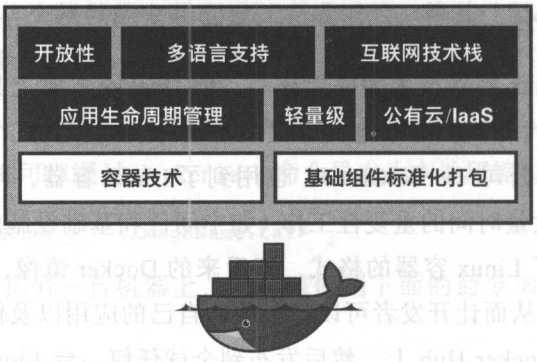


图 4-3 Docker 的关键特性

2）多语言支持。Docker 本身只是一个基础设施，并没有定义约束开发框架，也不提供面向业务的 API 接口，因此可以支持任意语言开发的应用。

3）互联网技术栈。Docker 一开始就是面向互联网开发的，并将互联网领域中常见的中间件都 Docker 化了。

4）应用生命周期管理。Docker 将软件开发的生命周期延长了，原来的开发周期到程序编译完成就结束了，引入 Docker 以后，可以扩展到应用打包镜像、实现自动部署和升级的整个过程。

5）轻量级。这主要是指相对于类似 OpenStack 的虚拟机技术。由于容器本身是轻量级的，容器共享主机的 Linux 内核，资源占有少，因此一个主机可以很快启动几百个容器，并且保持高速运行。

6）公有云 /IaaS。在国内，腾讯云一开始的定位就是 Docker 云。云设施从本质上来说是为支持应用服务开发的，由于容器云更便宜、更方便，所以未来它将是一个新的主流。除了腾讯云，目前国内还有好多小公司都在部署容器云。随着 Docker 之上的 Kubernetes 提供了先进的分布式微服务架构框架，未来会有更多的应用被改造并迁移到 Docker 环境中。

7）容器技术和基础组件标准化打包。严格来说，Docker 只有两件事是它自己做的，即容器标准化和镜像打包。容器技术和虚拟机技术基本上是并行开始的，但容器技术一开始是一个专有技术，是一套 API（LXC），没有面向个人，用户无法直接创建一个容器来使用，但是 Docker 将 LXC API 封装以后，将其标准化了，从而将容器技术这个只有少数人知道和掌握的高级技术变成普通用户可以使用的大众技术。为了让容器技术标准化和普及化，Docker 创新设计了镜像打包技术，把容器所有依赖的环境都打成一个标准包，将底层库、程序、配置文件等全部打成一个包并制作成二进制镜像文件后，就可以随意搬迁到任意服务器上或公有云上。

所以说，镜像技术是 Docker 最大的创新技术。

4.1.4 Docker 的口号

Docker 的口号很像 Java 的口号：“任何程序只要‘Build’一次，就可以被搬迁到任何环境中运行”，这个口号很有鼓动性。Docker 的口号有以下 3 个关键词，如图 4-4 所示。

1）Build：大多数情况下，Build 由运维工程师负责，将应用程序的安装过程固化为一个可以移植和复用的 Docker 镜像，这个过程是通过 Dockfile 以及相关的 docker build 命令来完成的。实际上，这是在 Linux 系统中安装和配置某个软件的过程，最终输出的就是一个类似光盘镜像的二进制文件。

2）Ship：Build 完成的镜像可以上传到 Docker Hub 存储，并被任意一台联网的计算机下载到本地，这个过程就是“Ship”。目前所有的镜像都存储在 Docker 官方维护的 Docker Hub 镜像仓库中，在国内如果网速较慢，可以到国内的时速云上查找是否有克隆的镜像可以下载。

3）Run：采用 docker run 命令启动一个容器的过程。虽然官方宣传 Docker 可以运行在任何环境中，但实际上 Docker 只能运行在高版本的 64 位 Linux 系统中。即使如此，也较 Java 的兼容性好，Java 希望任何设备都能兼容，结果反而是不兼容的问题频发。

图 4-4 Docker 的口号

4.1.5 Docker 正在成为当年的 Java

身披容器技术光环的 Docker 成为抗衡虚拟机技术的新秀，得到了众多大公司的垂青。Docker 之所以能够取得成功，除了自身强大的功能以及先进的技术等关键因素之外，也与其巧妙的销售宣传策略有关。最初 Docker 捕获了 IT 世界里众多基层群众的心，包括一线的程序员和运维工程师，之后不断渗透进一些小的 IT 公司，最终因为广泛的群众基础而引发了众多 IT 巨头的关注和投资，逐渐发展成为一个庞大的“帝国”。

Docker 兴起的时候正是 OpenStack 最鼎盛的时期，但后来的结果是 OpenStack 快速没落，而 Docker 则不断扩大地盘，这是为什么呢？最主要的原因是“简单”。众所周知，即使是对于一个非常熟练的同时懂网络的运维工程师而言，要安装一个 3 节点的 OpenStack 环境，最少也要用 3 天时间才能摸索成功，安装成功后又要花费很多时间学习它的功能，这些功能都比较复杂，如创建网络、创建虚拟机、挂接磁盘、虚拟机镜像制作等，与 Docker 的小巧、简单、灵活相比，OpenStack 简直就像是一只恐龙，所以注定会败落。容器技术具有虚拟机无法替代的优势，谷歌则是这方面的专家，谷歌在十几年的时间里一直专注于容器技术的开发和使用，并且部署了世界上最大规模的容器环境，包括几十万台物

理机。

一个技术如果背后有有影响力的公司推动，那么这个技术无疑是有前途的。虽然Docker最初是Docker公司开源的产品，但是现在Docker的主导权已经掌握在谷歌手里了。

图4-5是谷歌掌控Docker的路线图。2015年，谷歌发起了容器标准之争，当时CoreOS与谷歌拉拢了很多大公司加入了容器标准之争的战役，最终在Linux基金会的主持下，Docker、CoreOS、谷歌、红帽、Oracle、Microsoft、EMC、IBM、Cisco、VMware等公司一起发起了开放容器计划（OCI），这个名单里，Docker公司是最小的公司，而其他公司都一边倒地站在了谷歌的那一边，也就是说Docker（容器技术）的发展决策权从此被谷歌掌控了。OCI规范的诞生，意味着以后会有更多类似Docker的容器产品出现，大家都遵循统一的OCI标准，这样整个容器领域就形成了一个良好的生态圈，每个企业都可以参与，推动容器技术发展壮大。2016年4月Docker发布1.11版，是第一个兼容OCI标准的发行版，包括软件巨头、IT服务巨头、网络巨头、虚拟机巨头，软硬件公司全部向Docker靠拢，所以Docker的影响力越来越大，正在超越当年的Java。

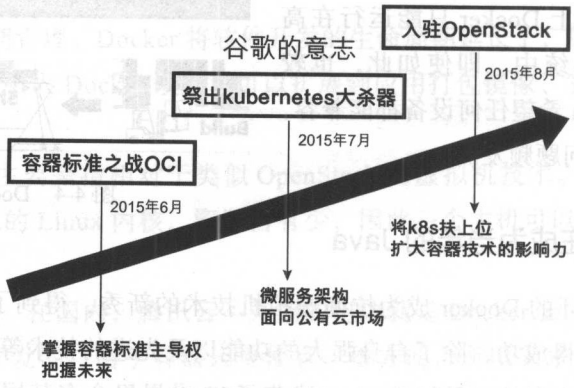


图4-5 Docker背后的力量

掌握了Docker控制权的谷歌随后又拿出了“大杀器”——Kubernetes。这是一个非常先进的微服务架构平台，具体先进在哪里？原来我们做的很多软件都希望能够实现监控、运维、修复这个闭环流程的自动化，即系统自动发现故障并尝试自动恢复，但是多年没有一个软件能够实现以上机制。Kubernetes则真正做到了这一点，因此一经推出，就吸引了众多软件厂商，红帽甚至直接放弃了OpenShift研发所积累的两代经验，第三个版本的底层就直接迁移到Kubernetes框架上了。

那么，谷歌为什么要开源这么好的技术呢？背后的原因很简单，大家都看到公有云是未来的趋势，越来越多的软件部署到了公有云上，而亚马逊在公有云市场上没有敌手。同为巨头的谷歌在公有云领域经营许久，竟然没有收获什么好处，所以他希望改变这一不利

局面，于是开源出 Kubernetes 并希望更多的人选择使用谷歌云。因为 Kubernetes 是谷歌开发的，如果用户选用 Kubernetes 架构并且希望托管到公有云，首先考虑的就是谷歌云，毕竟 Kubernetes 是谷歌开源的，肯定是自家的云平台支持得更好，这就是谷歌的“私心”，但不可否认的是 Kubernetes 的确对 IT 事业有巨大的贡献。

Docker 与 Kubernetes 的关系是什么样的呢？如图 4-6 所示，Docker 是基础设施，相当于 JVM，提供了一个容器运行时的环境；而 Kubernetes 类似 J2EE Server，提供了标准化的微服务架构。Kubernetes 自带负载均衡、命名服务和编排，并且自动运维。Kubernetes 把原来一直想解决的各种分布式的问题非常简单地解决了。

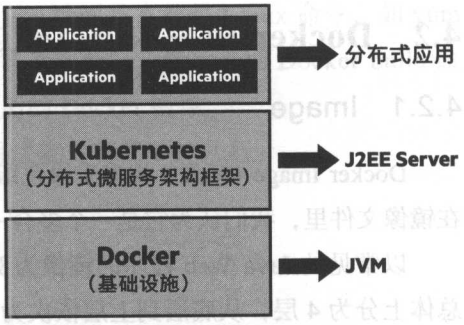


图 4-6 Docker 与 Kubernetes

2015 年，谷歌入驻 OpenStack，但目标并不是投资 OpenStack，而是扶持 Docker 和 Kubernetes 上位。OpenStack 做了 IaaS 层的基础工作，提供用户管理、租户管理、资源分配等基础功能，经过多年的发展依旧占据着一定的市场，因此，让 OpenStack 支持 Docker 的决定对于 Docker 的发展很有帮助。从这一系列行动来看，谷歌希望控制和发展 Docker 和 Kubernetes 技术，最终达到与当初 J2EE 一样的影响力。

Docker 的影响力越来越大，所有操作系统巨头包括一直不愿意屈服的微软，现在也已经“臣服”了。Docker 公司最近发布的一个声明表示，微软最强势的操作系统部门从来没有为开源做过贡献，但破天荒地第一次为 Docker 开源做了贡献，以支持 Docker 在 Windows 上运行。我们可以大胆地预测，未来 Windows 可能是双核的：一个 Windows 内核，一个 Linux 内核。有可能将来双核也会消失，变成一个内核（Linux）。

4.1.6 Docker 的部署环境要求

官方建议 Docker 的部署环境是 64 位 Linux，内核版本为 3.8.0 以上。Linux 是最重要的操作系统之一，学习 Docker 的时候，建议笔记本式计算机安装一个虚拟机环境，并且采用 CentOS 7.2 操作系统，这是因为 Linux 的“老大”是 Red Hat，企业环境多是 Red Hat/CentOS 系列的操作系统。

考虑 Docker 的影响力和发展速度，“Docker Ready”可能成为公有云虚拟机的一个标配特性，未来可能很多公有云提供的虚拟机都号称“Docker Ready”，或者具备一键就能安装 Docker 环境的能力。由于 Docker 隔离了底层的操作系统环境，它让我们对操作系统的依赖越来越小，我们在用 Docker 的时候完全无须关注下面是什么操作系统，对于服务端程序来说，甚至可以认为 Docker 就是“操作系统”了，深度支持 Docker 成为操作系统要面

临的新决策，于是红帽公司发布了下一代容器化的操作系统——Red Hat Atomic，Red Hat Atomic 内置了 Docker 以及 Kubernetes 服务，而微软的 Windows Server 2016 也开始深度支持 Docker，相信未来会有更多深度支持 Docker 的服务器操作系统诞生。

4.2 Docker 相关术语及概述

4.2.1 Image

Docker Image（镜像）相当于绿色版的二进制程序包，所有底层依赖的第三方软件都包含在镜像文件里，我们认为它是一个多合一（all in one）的光盘镜像，无须安装就可以直接运行。

以常见的 Java Web 程序的镜像为例说明 Docker 镜像的特点，如图 4-7 所示，这个镜像总体上分为 4 层，从底层到上层依次为：

- Linux 外围文件和程序层，可以理解为 CentOS 或者 Ubuntu 的“壳”，这一层的文件基本上包括 /root 目录、/etc 目录、/opt 目录，以及一些基础 Linux 程序文件，但不包括 Linux 内核程序和文件，因为容器共享主机上的 Linux 内核。其实所有 Linux 发行版本的内核是一样的，只是外面加的“壳”各自不同，不同的壳使用不同的文件系统，程序包和运行工具不一样。
- JDK 层，这部分是对应的 Open JDK 或者 Oracle JDK 程序，不同的镜像会安装不同的 JDK 版本。
- Tomcat 层，这一层安装好了 Apache Tomcat 程序的镜像层。
- User App 层，在 Tomcat 层里的 Webapp 目录下添加了我们所开发的 Java Web 程序包，从而在 Tomcat 启动的时候，就加载了我们的应用。

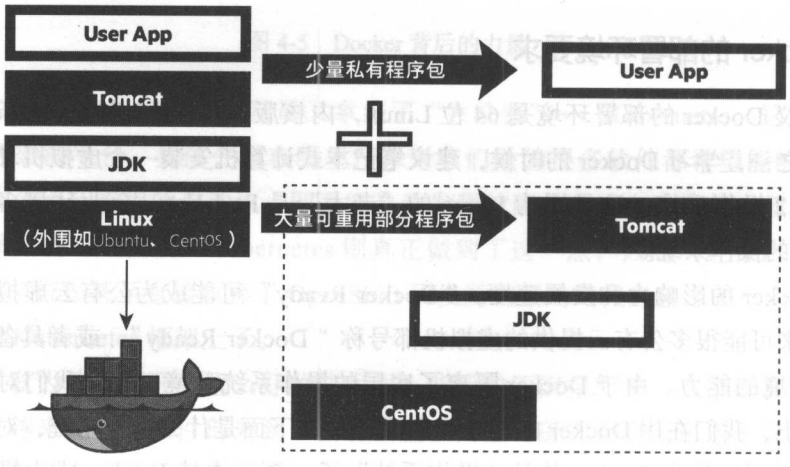


图 4-7 镜像

从上面的结构可以清晰地看出 Docker Image 的最大特点——分层继承，即上层镜像继承下层镜像，模仿了面向对象编程的“继承”思想，可以“继承”某个已有镜像，在此基础上增加自己的新内容，而不是一起从起点开始，这种设计使得 Docker 镜像的制作过程变得更加简单。

镜像制作的过程也不神秘，即编写一个 Dockerfile，里面基本都是 Linux 命令，如 yum install 安装软件、修改配置文件、复制文件到镜像中等基本命令，然后采用 Docker build 命令执行上述 Dockerfile 的操作，最后就产生了一个可用的 Docker 镜像。

例如，上面这个 Java Web App 的 Dockerfile 的写法就只有简单的几行：

```
FROM Tomcat
MAINTAINER bestme <bestme@hpe.com>
ADD demo /usr/local/Tomcat/webapps/demo
```

那么 Docker 镜像的实质是什么？按照 Docker 的做法，我们把安装配置过程写成一个 Dockerfile 文件，然后通过 Docker build 命令执行其中的指令，最后产生一个永久性的镜像，这个镜像可以在任何地方运行，不再耗费安装时间，节省了大量人力成本。因此，Docker 镜像的实质是指把 Linux 上某个应用的安装和配置活动提前“固化”并且自动化和标准化的过程，即把原来重复 *N* 次的人工安装配置过程，提前“固化”并“标准化”，图 4-8 直观地描述了这个结论。

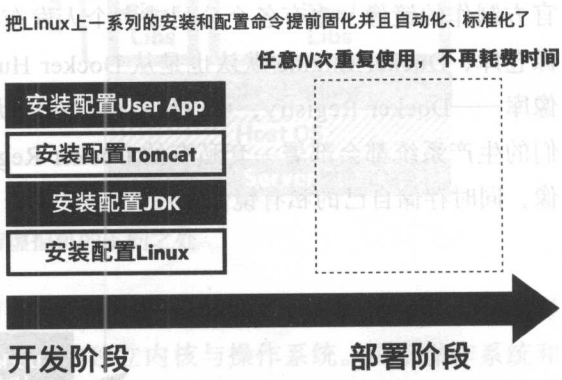


图 4-8 镜像的实质

此外，以前我们不知道一个程序的具体安装细节，但现在有了 Dockerfile 文件，每个人都能知道安装了什么软件，软件的版本是什么、在什么目录下，修改了什么配置，执行了什么命令，命令的参数又有哪些，所以 Docker 也强迫我们在安装、配置软件的过程中要尽量做到符合业界标准，做到规范化、标准化。

另外需要强调的是，Docker 镜像的要求是不可变的，要求制作镜像的时候，所必需的参数要通过环境变量的方式传递到容器内部，也就是说不能用某个镜像启动一个容器，然后进入容器里手动修改配置文件后重启容器，这对程序有要求，原来改造程序从配置文件读取启动参数，现在则需要从环境变量中去获取配置。Docker 镜像的这个特点避免了在部署过程中改文件导致的难以排查的低级错误。如果原来的镜像有问题，我们不用去修改原有的容器，而是打一个新镜像，并用新镜像重新创建一个新的容器。

4.2.2 Docker Registry

我们制作好的 Docker 镜像存放在哪里才能方便以后的多机器部署要求呢？答案是 Docker 镜像仓库（Docker Registry）。

Docker Registry 是 Docker 镜像的集中托管地，它完全参考了 Git Hub 的思想，具有版本管理功能，不同的是 Git Hub 托管的是源码，而 Docker Registry 托管的是二进制镜像文件。放入 Docker Registry 里面的每一个镜像都有一个 Tag（标签），可以将 Tag 看作镜像的版本，“镜像名称 +Tag”形成了镜像的唯一标识，因此同一个镜像可以在 Docker Registry 中保持多个版本，有了 Docker Registry，容器升级或回退到某个历史版本都很容易实现。

如图 4-9 所示，我们使用的 Docker Registry 其实有两种。一种是 Docker 官方维护的全球最大的镜像仓库——Docker Hub，这里存储了全球所有公开的 Docker 镜像，既有 Docker 官方制作的镜像，也有各个公司或者个人发布的镜像，任何联网的计算机都能访问这个镜像仓库，Docker Engine 默认也是从 Docker Hub 上拉取镜像的。另外一种是公司的私有镜像库——Docker Registry，可以用容器方式启动并供公司内部的机器所使用和访问，通常我们的生产系统都会部署一套私有的 Docker Registry，以便在网络隔离的情况下安全访问镜像，同时存储自己的私有镜像文件。

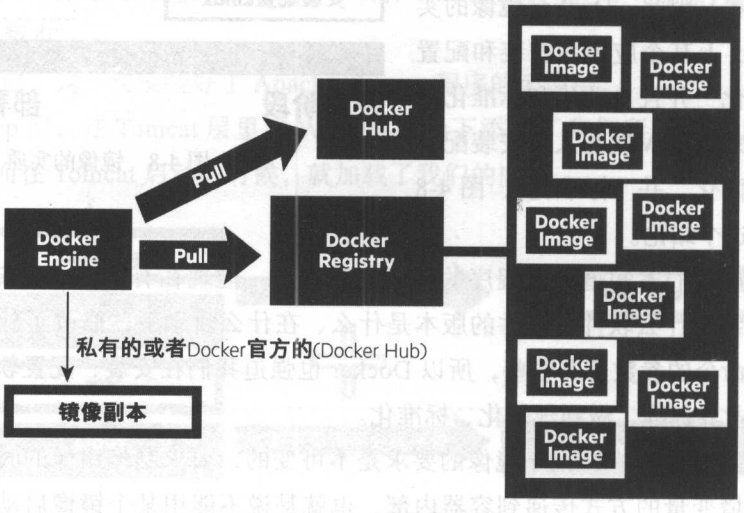


图 4-9 镜像仓库

4.2.3 Container

有了 Docker 镜像，我们就可使用这个镜像启动一个 Docker 容器（Container）了，

Docker 镜像类似光盘，而 Docker 容器就是从这个光盘启动的一个“迷你虚拟机”。

那么 Docker 容器与传统虚拟机有何不同？图 4-10 总结了容器与虚拟机的不同。

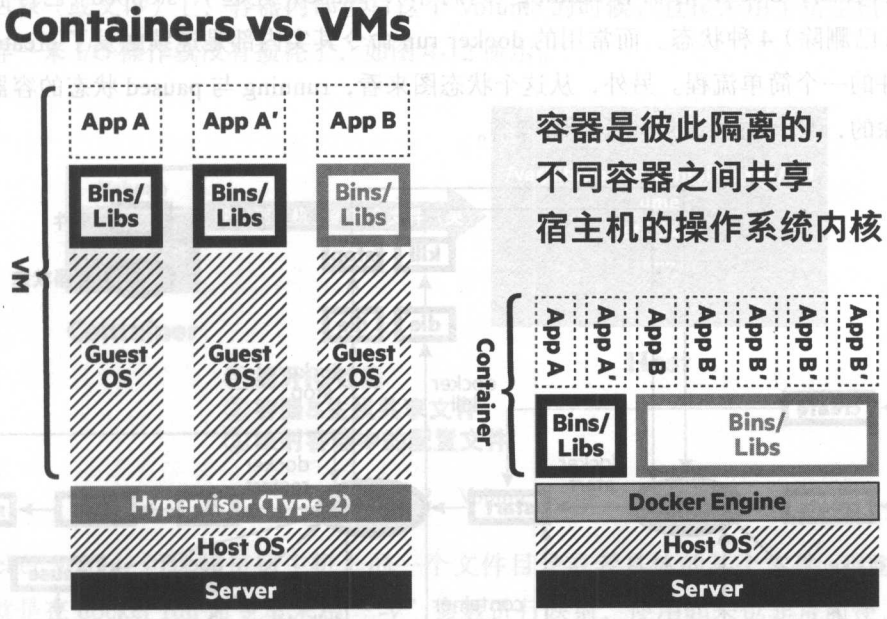


图 4-10 容器与虚拟机的不同之处

首先，一个物理机上的每台虚拟机都拥有一个完整的操作系统；但容器没有自己的操作系统，所有容器共享宿主机的内核，没有自己的独立内核与操作系统。由于操作系统和内核本身要占用大量的系统资源，所以虚拟机浪费了不少硬件资源；而容器共享同一个内核，所以容器更加节省物理资源，损耗小，因而更加轻量级。

其次，由于有独立的操作系统，所以每个虚拟机之间是完全隔离的，一个虚拟机宕机了不会影响其他虚拟机的正常运行，所以，虚拟机系统的整体稳定性非常高。而容器共享内核，因此当宿主机内核崩溃时，会导致所有容器一起烟消云散。如果我们要求非常高的整体稳定性，目前虚拟机是比容器好的一个选择。

传统的操作系统是为物理机而不是为虚拟机设计的，物理机的操作系统已经完成了一些基本工作，虚拟机的操作系统没必要再完整模仿物理机的操作系统，如果为虚拟机专门设计一套全新的操作系统，会更加节省资源，而且速度会更快。因此虚拟化领域的资深专家、KVM 之父 Avi Kivity 后来专门设计了一套全新的虚拟机操作系统——OSv。这是一个非 Linux 的新的操作系统，并且达到了 Docker 的特性，只需不到 1s 就可以启动，内含 JDK，并且重新写了很多重要的中间件。在 OSv 上运行的虚拟机要比普通虚拟机快很多倍，而且提供了 REST 方式的配置接口，可以实现程序化自动控制，未来操作系统特别是云上

的操作系统由程序来控制也将成为一个趋势。

Docker 容器类似虚拟机，是有生命周期的，图 4-11 是 Docker 容器的生命周期。从图中可以看到，Docker 容器有 running（运行中）、paused(挂起)、stopped（已停止）以及 deleted（已删除）4 种状态。而常用的 docker run 命令其实内部是连续触发了 create 与 start 两个事件的一个简单流程。另外，从这个状态图来看，running 与 paused 状态的容器是不能直接删除的，只有先 stop 后才能删除容器。

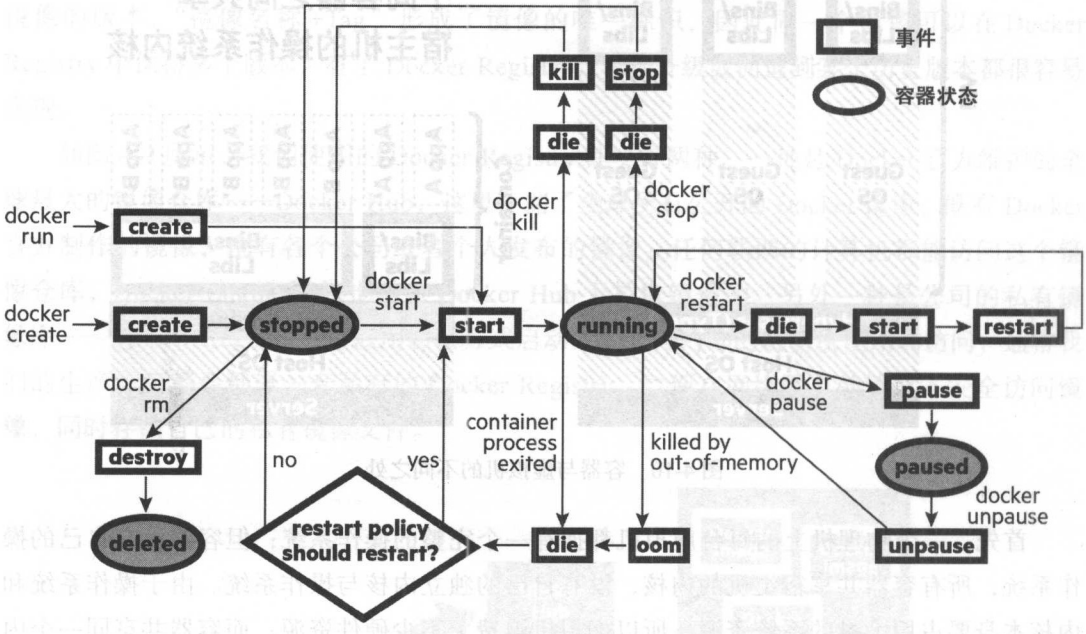


图 4-11　Docker 容器的生命周期

Docker 提供的容器生命周期管理功能对于微服务架构很有意义，每个微服务对象都有定义过程、启动过程，如果某个微服务产了问题，可以先挂起，待故障解决后再恢复运行或者删除实例。

4.2.4　Volume

Docker 镜像是分层文件系统，可以认为分层文件系统所管理的文件列表存储在一个特殊的“压缩包”里。利用 Docker 镜像产生的容器则使用了一种特殊的分层文件管理系统，它的作用是让访问分层文件系统的进程看到一个常规的“扁平”文件目录，而不是背后的“压缩包”。虽然 Docker 特有的分层文件管理系统解决了进程透明访问分层文件的问题，但带来的代价是 I/O 操作性能的降低。此外，由于容器内部的文件系统是完全隔离的，因此

在宿主机上是无法直接看到和访问容器内部文件的，为了解决这两个问题，Docker 引入了 Volume（挂接卷）的概念，允许把宿主机上的某个文件目录（或文件）映射到容器内部的某个指定目录（或文件）上，容器内部读写这个 Volume 的时候，直接采用了宿主机的 I/O 机制，这样一来 I/O 操作就没有损耗了，如图 4-12 所示。

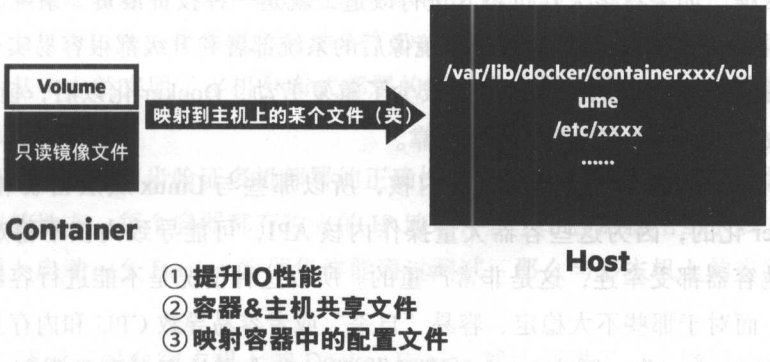


图 4-12 Volume

Docker Volume 可以挂接宿主机上的一个文件目录或者具体的某个文件，用法也非常简单，就是在 docker run 命令里采用“-v”参数进行映射，使用起来也非常简便，Docker Volume 具有以下几个常规用途。

1）容器与主机共享文件，通过 Volume 的方式让容器中的某个目录访问主机，实现文件共享。

2）将容器中的复杂配置文件放到宿主机上，通过 Volume 方式挂接到容器内，实现配置文件与镜像分离的目的，某些容器内的配置文件非常复杂，如果用环境变量传递参数，然后写一个定制脚本生成环境变量，会比较费时费力，还容易出错。

3）共享存储，Docker Volume 采用了插件机制，比较容易扩展，因此 Kubernetes 等第三方系统提供了更多的选择，如 NFS 共享存储，以及 GlusterFS 分布式文件系统，解决了分布式情况下的文件共享问题。

4.3 如何用 Docker 改造传统项目

4.3.1 哪些应用适合 Docker 化改造

首先，哪些应用适合 Docker 化改造？我们认为符合以下条件的应用是适合进行 Docker 化改造的。

1）频繁修改和升级的系统。

2）不太稳定，容易死机或者导致 CPU、内存等过度消耗的应用。

3）大量使用开源技术和中间件的系统。

4）其主要目的是用作演示的应用。

首先，很少变动的系统不适合进行 Docker 化改造，因为很少变动的系统意味着没有什么业务需求支撑，如果花很大代价将其进行改造，就是一种投资浪费。系统改动升级越频繁，Docker 化的价值越大，因为 Docker 镜像后的系统部署和升级都很容易实现标准化和自动化，大大提升了项目组的工作效率，减少了重复劳动，Docker 化以后，镜像都有版本，因此升级失败后的回滚也变得更容易和可靠。

其次，Docker 容器因为共享 Linux 内核，所以那些与 Linux 底层密切相关的应用是不适合 Docker 化的，因为这些容器大量操作内核 API，可能导致内核不稳定，如内核崩溃，此时其他容器都受牵连，这是非常严重的。所以这种情况是不能进行容器化的，必须要独立出去。而对于那些不太稳定、容易“自杀”或者容易导致 CPU 和内存过度消耗的应用，则很适合 Docker 化，因为 Docker 可以做 CPU 和内存的资源配额限制，例如，一个 Docker 容器只分配 1GB 的内存，则此容器内的进程如果申请超过 1GB 的内存，就可能会被 Docker Engine“杀死”，如果该容器限定了使用 0.5 个 CPU，那它永远不会得到超过 0.5 个 CPU 的使用量，因此 Docker 化以后，这种过度消耗资源的进程会被约束，不会导致系统问题。另外，如果某个进程不稳定并且容易“自杀”，则可以在容器化的时候设置它的启动策略为“ restart=always”，这样如果 Docker Engine 发现这个容器“死了”，就会自动尝试重启这个容器。

最后，其主要目的是用作演示的应用以及项目 POC 也都适合 Docker 化。因为 Docker 比较轻量，很容易用 Docker 容器部署一个集群应用，这个集群应用只需要少量资源，如在高配置的笔记本式计算机上就能流畅演示。此外，如果演示者本身并不是很懂技术，相对于虚拟机等复杂的技术，Docker 的使用还是很容易快速掌握的，稍微培训一下，很多售前和销售人员就能正确启动一套 Docker 的演示集群。此外，每次创建的 Docker 容器都是全新的、没有历史数据的，因此每次演示都能确保系统是正常可靠的。

4.3.2 Docker 化改造传统应用的流程

Docker 化改造传统应用的关键步骤如下。

1）评估代价与可行性——评估改造的可行性及资源投入等。

2）改造方案——包括如何升级改造、对代码进行修订的指导性意见等。

3）代码修订——对前期涉及的原有代码进行修订，包括一些接口设计等。

4）制作镜像——实施阶段，制作镜像。

5）单机验证测试——对镜像做单机验证。

6）多机部署——多机环境的正常部署和应用的正常启动。

首先根据系统的特点判断是否适合 Docker 化改造，如适合改造，则开始制定改造方案，改造方案会涉及系统镜像的组成、镜像的参数、镜像的启动方式以及源码改造点等基本问题。接下来就是改造环节了，即改造代码以符合 Docker 的要求，然后制作镜像。最后是验证阶段，由于可以在一个单机上启动多个 Docker 容器进行联网调测，所以单机测试验证通过以后，基本上就表明了多机分布式部署的正确性，最后进行多机部署测试，以及生产环境的上线过程。

关于单机部署能够初步验证多机部署的正确性，这里再解释一下，这是因为 Docker 与虚拟机有相似的地方，每个容器都有独立的 IP 地址，多一个容器就相当于多一个主机。如果在一台机器上启动一个 Docker 容器集群能通过测试，那么与多主机上的容器集群是没有本质区别的。

如图 4-13 所示，每台宿主机上的 Docker Engine 默认创建了一个网段为 172.17.0.0/16 的 B 类地址的虚拟网络，这个虚拟网络的网桥是 docker0，docker0 这个网桥是通过 Linux 的 Bridge 来实现的，Bridge 是 Linux 上用来实现 TCP/IP 二层通信的基础（虚拟）设备，与真实世界中的交换机设备类似，一个宿主机上的每个 Docker 容器都被连接到了 docker0 网桥上，并且被分配了 172.17.0.0/16 网段中的某个 IP 地址，因此它们之间可以通过 docker0 网桥实现本地通信，而当它们要访问外网的时候，则可以通过 docker0 网桥的网关地址转发到宿主机的物理网卡上，从而实现容器到外网的网络通信。而不同宿主机上的 Docker 容器要实现互通则有很多成熟方案可选，当前基于 VXLAN 的 Overlay 网络是最为标准的一种方案，这也是 Docker 1.13 默认内置的网络方案。

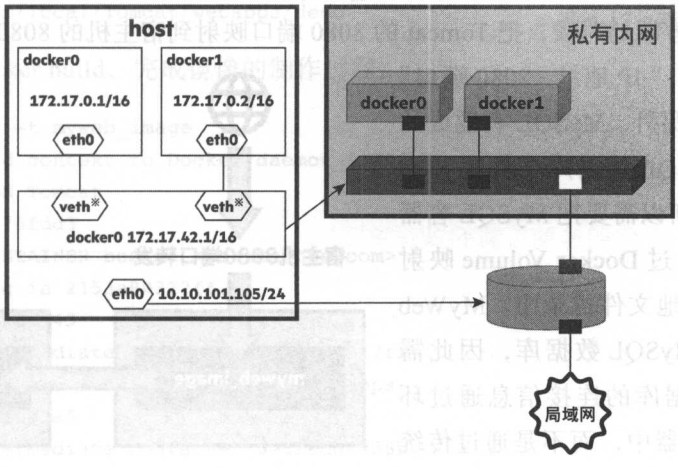


图 4-13 Docker 容器网络示意图

如果其他主机要访问某个宿主机上某个容器的端口，只能通过端口映射方式来实现，即容器内的 TCP 端口绑定到宿主机的某个端口上，当宿主机收到 TCP/IP 请求的时候，就转发到容器对应的端口，这就是 Docker run 命令中的参数 -p 的作用，比如 -p 80:8080 表示将容器中的 8080 端口映射到主机的 80 端口上。

4.3.3 Docker 化改造案例

这里以一个简单的案例来说明如何完成旧系统的 Docker 化改造过程。

图 4-14 是一个传统并且有代表性的 Java Web 系统，这个系统是运行在 Tomcat 上的一个 Web 应用，通过 JDBC 访问 MySQL 数据库并且采用 JSP 展示数据。

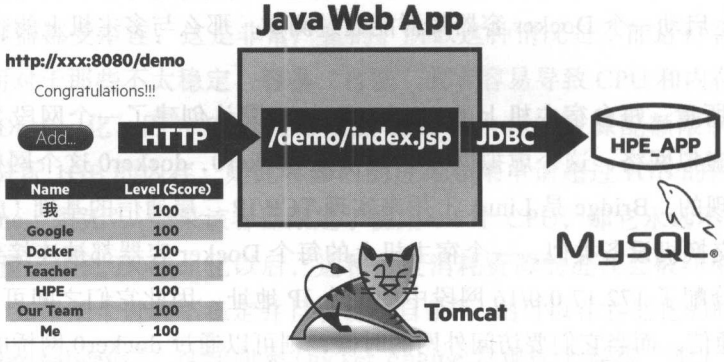


图 4-14 Docker 化改造的例子

其实 Java 程序是很容易 Docker 化的，上述程序改造方案如下。

Web 应用可以做成一个新镜像（myweb_image），这个镜像以 Tomcat 镜像为基础，增加应用代码，启动容器的时候，把 Tomcat 的 8080 端口映射到宿主机的 8080 端口，随后就可以通过宿主机的“IP 地址 +8080 端口”来访问应用了。另外，MySQL 有现成的可用，由于 MySQL 数据库文件需要高性能 I/O 读写，所以需要把 MySQL 容器的数据库文件通过 Docker Volume 映射到宿主机上的本地文件目录中。MyWeb 容器需要访问 MySQL 数据库，因此需要把 MySQL 数据库的连接信息通过环境变量传递到容器中，而不是通过传统的读取本地配置文件的方式，这是程序

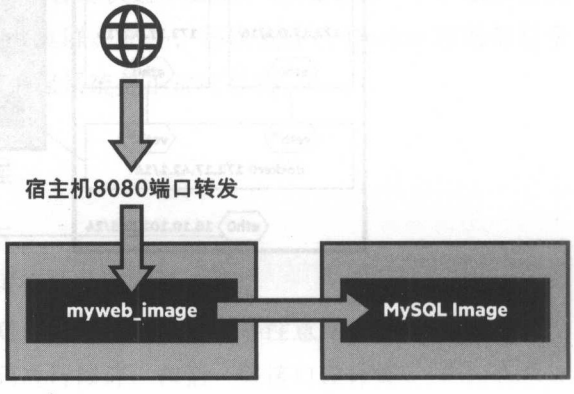


图 4-15 整体改造方案

中唯一要改造的地方。我们通常看到的 Docker 化过程中代码的改造工作量只是很小的一部分，图 4-15 给出了整体改造方案。

Java 本身也提供从系统环境变量中获取的配置参数，所以代码的修订很简单，如图 4-16 所示。

demo/index.jsp

```
<body  align="center">
  <%
java.sql.Connection conn=null;
java.lang.String strConn;
java.sql.Statement stmt=null;
java.sql.ResultSet rs=null;
Class.forName("com.mysql.jdbc.Driver").newInstance();
try{
      Class.forName("com.mysql.jdbc.Driver");
       String ip=System.getenv("MYSQL_IP");
        ip=(ip==null)?"localhost":ip;
      System.out.println("Connecting to database...");
      conn = java.sql.DriverManager.getConnection("jdbc:mysql://"+ip+":3306", "root","123456");
      stmt = conn.createStatement();
```

从环境变量中获取MySQL连接地址

图 4-16 代码修订

接下来，我们看看 myweb_image 这个新镜像的结构，如图 4-17 所示。

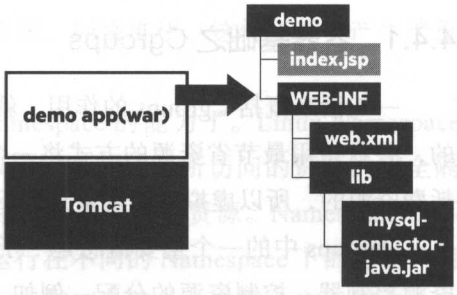


图 4-17 镜像结构

下面是新镜像的 Dockerfile 文件，由于新镜像依赖 Tomcat，只要在 Tomcat 的发布目录下添加 demo 程序包即可，所以这个镜像特别简单：

```
FROM Tomcat
MAINTAINER bestme <bestme@hpe.com>
ADD demo /usr/local/Tomcat/webapps/demo
```

最后运行 docker build，完成镜像的制作过程：

```
docker build -t myweb_image  .
Sending build context to Docker daemon 1.006 MB
Step 1 : FROM Tomcat
 ---> 7f050c76fdd3
Step 2 : MAINTAINER bestme <bestme@hpe.com>
 ---> Running in 2153a90317ff
 ---> cc6e10b0e143
Removing intermediate container 2153a90317ff
Step 3 : ADD demo /usr/local/Tomcat/webapps
 ---> 0cd8901bc1a6
Removing intermediate container 6e3feeb1938b
Successfully built 0cd8901bc1a6
```

最后是部署与测试，如图 4-18 所示。

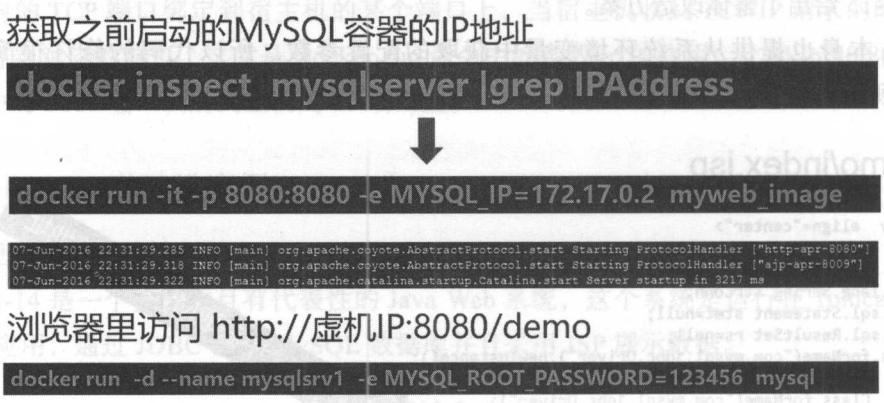


图 4-18 部署与测试

4.4 Docker 高级进阶

4.4.1 容器基础之 Cgroups

一句话来概括 Cgroups 的作用，就是限制进程所使用的资源配额。Cgroups 是谷歌发明的，谷歌想用最节省资源的方式将一个主机的资源被多个进程分享复用，但虚拟机是非常耗费资源的，所以虚拟机的这条路走不通，于是谷歌发明了 Cgroups。

Cgroups 中的一个重要概念是“子系统”，每种子系统就是一个资源的分配器，也就是资源控制器，控制资源的分配。例如，CPU 子系统是控制 CPU 时间分配的，memory 子系统是控制内存分配的。只要在系统中创建一个 Cgroups 子系统节点，并将控制的属性写入，然后将需要控制的进程 ID 写入这个节点，就实现了目标进程某种资源限制的目标。可以用多种不同的 Cgroups 子系统来控制同一个进程，从而实现目标进程占用的 CPU、内存、I/O 等资源的全面限制。

如图 4-19 所示，Task1、Task2 两个进程被不同的 Cgroups 子系统所约束，而 Docker 容器也采用了类似做法来实现不同容器的资源配额限制，即为每个容器定义一个 Task Group，挂接不同的 Cgroups 资源控制器，从而让这个容器内的所有进程（Task）使用的资源总额不超过预设值。例如，使用 docker run 命令的时候增加参数：-m 100m 表示限制该容器能使用的最大内存为 100 MB。而 CPU 的限制比较复杂，有相对权重模式、CPU 时间百分比等几种方式可以选择。

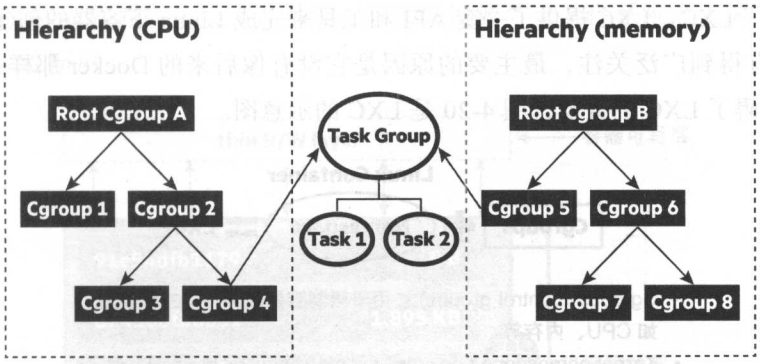


图 4-19 容器技术基础之 Cgroups

4.4.2 容器基础之 Namespace

虽然使用 Cgroups 可以把容器所用的资源配额进行限制，大家可以很好地共享主机资源，避免相互影响，但此时一个主机上的不同进程仍然可以相互“看到”彼此的存在。例如，A 进程可以“看到”B 进程，所以 A 进程可以调用操作系统的 API 把 B 进程“杀掉”。另外，A 进程与 B 进程仍然使用同样的宿主机文件系统、网络堆栈，这样仍然会产生严重的干扰问题。

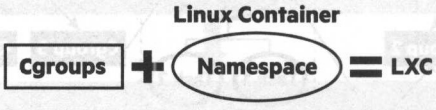
如何解决这个严重问题呢？这就要靠 Linux Namespace 的能力了。Linux Namespace 与 Cgroups 一样，属于 Linux 内核级的功能，它可以实现不同进程所访问的资源的完全隔离，包括网络栈、文件系统、进程编号、主机名、用户等所有关键资源。Namespace 对不同的容器进行隔离后的效果怎么样呢？简单来说，运行在不同的 Namespace 下的容器，拥有完全独立的进程编号，进程 ID 都是从 1 开始的，“看不到”其他的进程，容器拥有自己独立的主机名、独立的网络栈、独立的 IP 地址、独立的文件系统、独立的用户，也就是说 Namespace 让 Docker 容器成为真正的“容器”。

4.4.3 Docker 的容器原理

Cgroups 实现资源配额管理，Namespace 实现资源隔离，两者的结合相当于虚拟机。但容器还有一个与虚拟机显著不同的特点，即容器一定是与某个进程相关的，容器内部有一个用户指定的进程以前台模式运行，当这个用户进程停止时，容器就停止了，我们不可能启动一个没有前台进程运行的容器。

虽然 Cgroups 与 Namespace 是构成 Linux 容器的两大核心，但要创建一个真正可用的 Linux 容器，还是很复杂的一个专业技术活。早在 Docker 之前，就有一套知名的 Linux 容

器开源项目——LXC。LXC 提供了一套 API 和工具来完成 Linux 下容器的创建和管理功能，但 LXC 却没有得到广泛关注，最主要的原因是它没有像后来的 Docker 那样，提供标准化的镜像，这阻碍了 LXC 的普及，图 4-20 是 LXC 的示意图。



- Cgroups (control groups)：用于限制容器能使用的主机资源，如 CPU、内存等
- Kernel namespace (ipc、uts、fs、pid、network 和 user)：命名空间，使用不同容器具有不同的网络栈、主机名、进程编号等
- chroots (using pivot_root)：容器里的文件系统是一个完整的 Linux 系统，如有 /etc、/lib 等目录，可以通过 chroot 实现
- AppArmor and SELinux profiles：容器安全相关的技术，如是否允许访问主机网络和文件系统
- Seccomp policies：全称为 Secure computing mode，为“不可信的纯计算型代码”，提供一个“安全”的运行环境
- Kernel Capabilities：一些 Linux Kernel 的特性支持

图 4-20 Docker 与 LXC

如图 4-21 所示，Docker 最早也是使用了 LXC 的 API 来实现容器创建和管理工作的，此外 libvirt 这套被 OpenStack 所使用的 API 也支持创建 LXC 容器，还有 systemd-nspawn 也支持创建 Linux 容器，这些 API 或者 Lib 库都依赖 Linux 的一些内核技术，比如 Namespace、Cgroups、Netlink、Netfilter、SELinux、AppArmor 等。随着 Docker 的迅猛发展，LXC 已经不能满足需求，Docker 需要支持更多的平台而不仅仅局限于 Linux 本身，因此 Docker 需要一套新的容器接口规范和实现机制，这就是后来的 Libcontainer。Libcontainer 成熟后融入 Docker Engine 中成为替代 LXC 的容器 Driver。

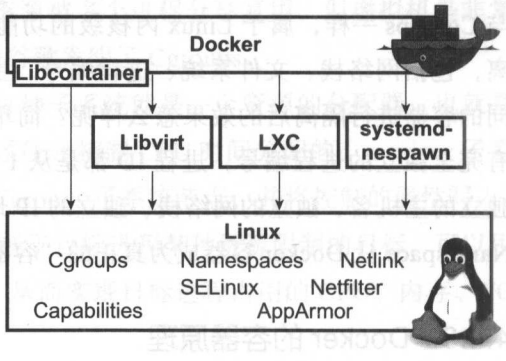


图 4-21 Docker 容器的常用 Driver

4.4.4 Docker 的分层镜像原理

分层镜像是 Docker 的另外一个核心特性。我们知道 Docker 镜像是分层结构，即由多层文件系统所组成，但从容器中进程的视角来看，这又是一个扁平化的文件系统，并且 Docker 的每一层镜像都是只读的，那么 Docker 容器又是如何做到在容器内读写文件的呢？

答案很简单，Docker 容器的分层文件系统在只读的镜像分层文件系统之上叠加了一层“可写层”，如图 4-22 所示。

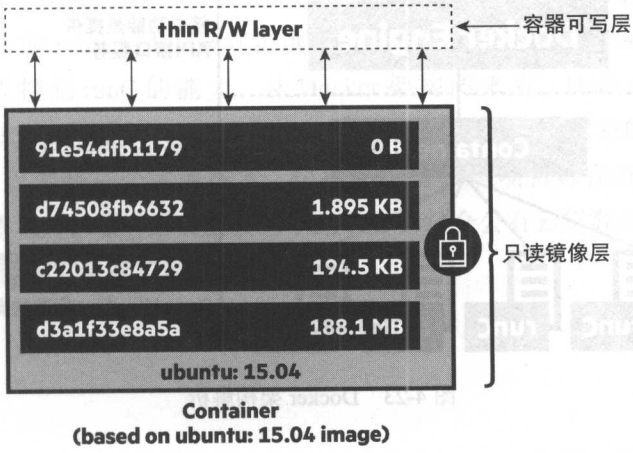


图 4-22 Docker 容器的分层文件系统

为了支持 Docker 镜像的特殊分层文件系统，Docker 最早设计了 AUFS（Another Union File System）分层文件系统，其主要特性是支持将多个目录挂载到同一个虚拟目录下，但 AUFS 是基于 Ubuntu 研发的，未能进入 Linux 内核里，从而对于非 Ubuntu 的 Linux 分发包，如 CentOS 就无法使用 AUFS 作为 Docker 的文件系统。于是，作为第二优先级的 DeviceMapper 就被作为分层镜像的实现方式。DeviceMapper 自 Linux 2.6 被引入内核后成为 Linux 最重要的一个技术，它在内核中支持逻辑卷管理的通用设备映射机制，为实现用于存储资源管理的设备驱动提供了一个高度模块化的内核架构，在比较长的一段时期内，Docker 采用了 DeviceMapper 来作为分层镜像和容器的存储实现方式。

Docker 1.12 之后，Docker 镜像存储的实现方式又“升级”了，默认采用了 OverlayFS 分层文件系统，它不同于 AUFS，是属于 Linux 内核支持的分层文件系统，被纳入 Linux 3.18 内核。此外，跟 AUFS 的多层不同的是 OverlayFS 只有两层：一个 Upper 文件系统和一个 Lower 文件系统，分别代表 Docker 的镜像层和容器层。当需要修改一个文件时，使用 Copy On Write 技术将文件从只读的 Lower 层复制到可写的 Upper 层进行修改，结果也保存在 Upper 层。

4.4.5 Docker 架构解析

Docker 采用了插件模式进行设计，很多重要模块都可以被扩展和替换，而总体上 Docker 的架构可以划分为以下几个部分，如图 4-23 所示。

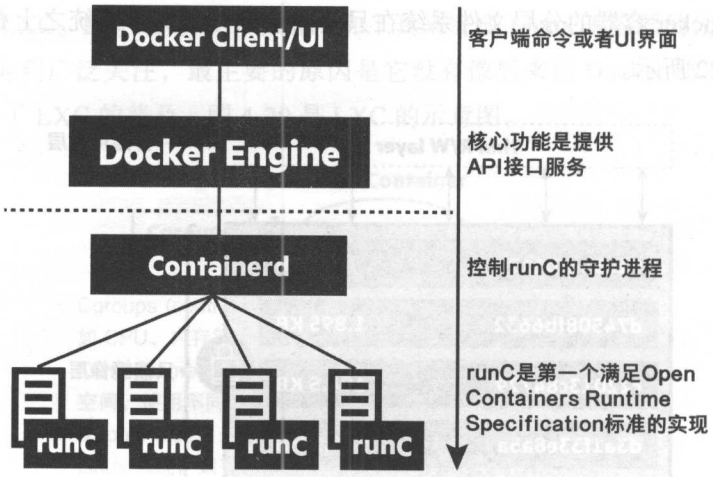


图 4-23 Docker 架构解析

从上往下看，第一层是 Docker 命令行工具，命令行工具通过 socket 方式调用 Docker Engine 提供的 API，这些 API 本质上都是 REST 接口，提供了镜像 Build、镜像上传、镜像下载、容器创建、容器生命周期管理、容器监控等各种功能。图 4-24 是 Docker Engine 提供的 REST API 的示意图。

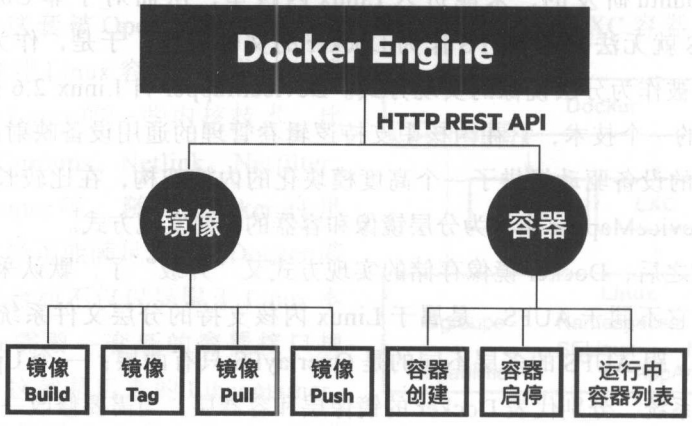


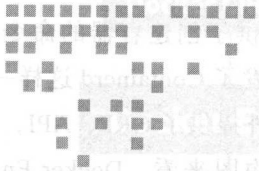
图 4-24 使用 Docker API 开发容器平台

目前 GitHub 上有不少 Docker Client API，其中包括 Java 实现的客户端 API，意味着我们完全可以以编程的方式来实现 Docker 命令行的所有功能。Kubernetes 也依赖 Docker Engine 提供的接口来创建和管理 Pod 对应的容器。

Docker Engine 以下的部分是 OCI 标准规范定义的组件，其中 runC（https://runc.io/）是 OCI 组织发布的标准化的通用轻量级容器运行环境，目前 runC 是一个命令行工具，可

以生成OCI规范的“标准容器”。runC最早是由Docker公司贡献的源码发展而来的，使用Libcontainer来驱动和构建容器实例。runC仅仅提供了创建容器的命令行工具，没有提供一套API供编程调用，所以后来Docker公司开发了Containerd这样一个新的项目，Containerd是用于控制runC的Linux守护进程，它对外提供了GRPC API，从而为Docker Engine提供了编程控制runC的能力，从Docker架构图来看，Docker Engine就是通过Containerd的API来控制底层的runC以及runC上运行的各种容器对象的。

目前，Docker已经是一个主流技术，包括ERP等传统项目都在尝试Docker化改造，系统Docker化改造之后，很容易实现公司本地服务器结合公有云资源的按需弹性扩展，当服务容量不够的时候可以快速扩容，非常便利。



Chapter 3 第 5 章

Kubernetes

在云计算时代，容器技术已经在各大 IT 公司广泛使用。谷歌公司的 GAE（Google App Engine）、微软公司的 Microsoft Azure、VMware 公司的 Cloud Foundry，以及国内新浪的 SAE（Sina App Engine）都采用了比虚拟机更小的颗粒度来运行应用。也就是说，在这些平台上，应用程序已经运行在类似容器的载体之上。早期开源的容器技术代表应该算是谷歌的 IMCTFY（Let Me Container That For You）了，可惜它中途“夭折”了。不过，这些容器技术大部分都长期停留在企业内部使用阶段，其便捷和高效的特性未能惠及广大开发人员和用户，直到 Docker 的问世。

Docker 的横空出世让沉寂已久的容器技术重回公众视野，并迅速成为业界的焦点。对于开发者和用户而言，Docker 将应用打包为更轻量级的镜像，运行在更细粒度的容器技术之上，使得容器具备了跨越不同计算平台快速部署和运行的能力，使应用能够满足变化越来越快的业务需求。伴随着近年来大型企业 IT 系统中持续增长的基础设施虚拟化比例，以 Docker 为代表的容器技术已经成为云计算领域中颠覆传统 IT 的又一新生力量。

2014 年，借着 Docker 掀起的容器化浪潮，谷歌迅速推出开源容器编排工具——Kubernetes，并在 2015 年 7 月推出 1.0 版本，占据了大规模容器集群管理领域的霸主地位。本章将介绍 Kubernetes 的概述、核心组件和关键概念，并借助一个完整实例演示一个真实的应用场景，还会讨论 Kubernetes 的高级特性。

5.1 Kubernetes 的背景与概述

5.1.1 谷歌保守了十几年的秘密武器——Borg 系统

在介绍 Kubernetes 之前，让我们先了解一下谷歌保守了十几年的秘密武器——Borg 系统。在过去的十几年间，谷歌用 Borg 来管理各种应用软件在其数据中心成千上万台服务器上的运行。谷歌从未公开过 Borg 系统的设计细节，而是将它视为自己的竞争优势隐藏起来。谷歌对于技术的态度常常是，经过几年的实践和发展，发表一篇描述该技术的论文。然后外界就会“山寨”这种技术，很多开源工具都是这样发展出来的，例如，MapReduce 催生了 Hadoop，BigTable 催生了一批 NoSQL 数据库。同样的事情也发生在 Borg 系统上。不过现在在云业务需求的推动下，谷歌正在改变它的战略，它分享出来的不仅仅只是一篇 Borg 系统的论文，而是把整个核心体系转变成了开源项目 Kubernetes。

Borg 系统的主要特性包括：①屏蔽资源管理和错误处理等细节，使用户关注于应用本身的开发工作；②提供高可用和高可靠的平台，同时对应用本身也支持高可用性和高可靠性；③在成千上万台机器上高效运行工作任务。截至 2015 年，谷歌在全球共拥有 36 个数据中心，有的数据中心拥有多达 7.5 万台服务器，总服务器的数量达到近 100 万台，在这些服务器上每周启动的容器数量超过了 20 亿个。如图 5-1 所示，Borg 承载着谷歌的各种业务系统，包括谷歌搜索引擎（Google Search）、邮件服务（Gmail）、文档协作服务（Google Docs）、谷歌地图（Google Maps）、大数据计算（MapReduce）、分布式数据存储（BigTable）等。

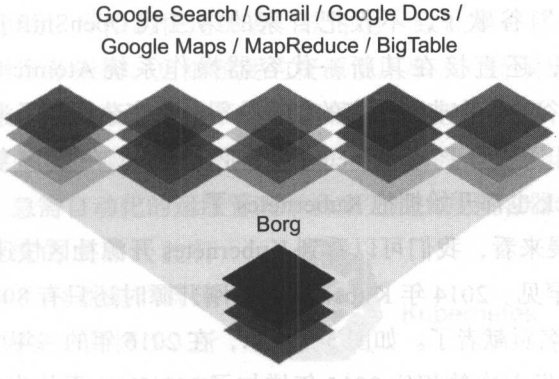


图 5-1 Borg 承载的谷歌业务系统

5.1.2 Kubernetes 的起源

Kubernetes 最初是在 2014 年由谷歌的 3 位工程师 Craig McLuckie、Joe Beda 和 Brendan Burns 一起构思的，他们都是谷歌云服务开发团队的成员。McLuckie 在把这个想法

告诉谷歌全球数据中心网络监管人 Urs Hölzle 时，Urs 并不喜欢这个想法，因为这会把谷歌的秘密武器作为开源技术拱手让人。但是在微软、VMware、EMC 等 IT 巨头纷纷拥抱开源的大趋势下，同时在 McLuckie、Beda 和 Burns 看来，Kubernetes 这个开源项目可以有力地推动开发者使用谷歌的云计算服务平台 GCE（Google Compute Engine），谷歌逐渐开始意识到开源环境推进云战略的重要性，哪怕把它最大的秘密部分开源化也在所不惜。

正如 Brendan Burns 所说，他和 Beda、McLuckie 都看到了有很多其他项目正在利用 Borg 及容器技术的思想，所以他们认为谷歌可以推进这件事情。他说："我们真的感觉到人们在将一块块碎片拼接时遇到很多困难，而我们有完整的拼图。我们有 10 年的经验，知道怎么把碎片拼到一起。"[⊖]

诚然，Kubernetes 并不是开源的 Borg，它没有 Borg 或 Omega（Borg 的继任者）那么复杂。但 Kubernetes 项目的目标是纠正 Borg、Omega 的错误，最终超越这两位"前辈"。在目前这个如此器重开源的时代，这也是谷歌参与竞争的最好方式。对于很多人来说，容器就是软件开发的未来，而谷歌现在正在加速这个未来的实现。

Kubernetes 这个名字来源于古希腊语，意思是舵手、导航员，也是 Cyber 的词源，Kubernetes 的目的是要让容器能用于企业的生产环境，使容器集群的配置标准化，让分布式应用的开发和部署更加容易。同时，谷歌也希望 Kubernetes 能够成为鼓励用户使用谷歌云服务的舵手。

从影响力来看，2015 年 Kubernetes 一经推出，各大 IT 巨头都纷纷跟进，包括红帽、微软、英特尔、Citrix 等。作为软件虚拟化领域的领导者之一，红帽在容器技术方面已经完全"跟从"谷歌了，不仅把自家的第三代 OpenShift 产品的架构底层换成了 Docker+Kubernetes，还直接在其新一代容器操作系统 Atomic 内集成了 Kubernetes。CoreOS 公司是一家和谷歌合作非常紧密的创业公司，这家公司除了为 Kubernetes 提供许多核心功能的开发之外，还致力于 Kubernetes 的商业化运作。在 2016 年，甚至老牌虚拟化厂商 VMware、OpenStack 也都开始拥抱 Kubernetes 了。

从过去两年的发展来看，我们可以看到 Kubernetes 开源社区快速壮大的过程，其火爆程度在开源社区相当罕见。2014 年 Kubernetes 刚刚开源时还只有 80 几个贡献者，到 2016 年年底已经有 1000 多名贡献者了。如图 5-2 所示，在 2016 年的一年中，Kubernetes 就经过了 34 次稳定版发布，发布次数相比 2015 年增加了 112.5%，平均发布间隔为 11 天，并且丝毫没有减慢的趋势。[⊖]

⊖ Google Made Its Secret Blueprint Public to Boost Its Cloud. https://www.wired.com/2015/06/google-kubernetes-says-future-cloud-computing/

⊖ Kubernetes A Year in Review. http://static.lwy.io/pdf/kubernetes_-_a_year_in_review_2016.pdf

图 5-2 Kubernetes 在 2016 年的发布次数

在 2015 年，谷歌还加入了 OpenStack 基金会，并且联合其他 20 家公司成立了开源组织云原生计算基金会 Cloud Native Computing Foundation（CNCF）。CNCF 的主要目标就是确保 Kubernetes 未来在任何基础设施（公有云、私有云、混合云和裸机）上都能良好地运行，并推动与合作伙伴社区共同开发容器管理工具集。Kubernetes 目前已经被全球多家大型企业应用到生产环境中，包括 eBay、Pearson、Wikimedia、Box、SAP、纽约时报、Yahoo、中国移动、网易、新浪 SAE 等公司，并正在被更多的企业用户接纳和使用。

5.1.3 Kubernetes 的核心特性

在介绍 Kubernetes 的核心特性之前，让我们先回顾一下容器技术的特点。容器技术具有许多优势，包括轻量级、应用的快速部署、软件的一致性和可移植性、以应用为核心、支持松耦合分布式的微服务架构、更细粒度的资源使用率、对持续集成和持续发布的更好支撑等。从开发到运维，人们渐渐适应了使用容器对软件进行包装、发布和部署。但随着数据中心服务器（主要是 X86 服务器）数量的增长、应用的微服务化，需要部署的容器种类和数量也越来越多，急需自动化的运维管理平台来完成大规模容器集群的发布和管理。

谷歌基于 Borg 系统的成熟经验打造的 Kubernetes，正是为容器集群管理提供的完整开源解决方案。如图 5-3 所示，Kubernetes 的主要特点包括：1）轻量级，类似于绿色软件，下载即可使用；2）便携性，Kubernetes 可以在公有云、私有云或混合云等任意环境上部署；3）可扩展性，Kubernetes 采用了插件化的设计架构，每个模块都是可插拔的，

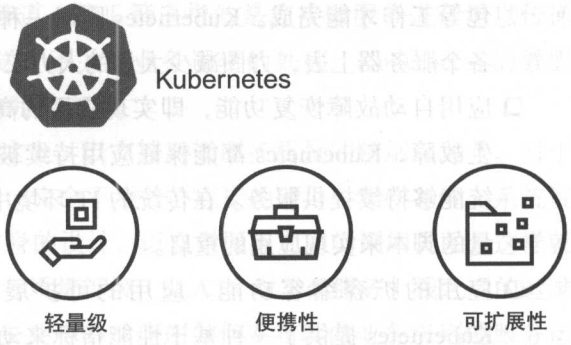


图 5-3 Kubernetes 的特点

也可以使用第三方插件进行替换。

Kubernetes 的功能非常丰富，从资源调度、容器发布、状态监控、弹性扩缩容、滚动更新、故障恢复，到服务发现、负载均衡等方方面面，可以实现分布式容器应用的大规模集群管理。同时，Kubernetes 为构建微服务系统提供了全套标准化的架构，通过 Pod、RC、Service、Label 等核心概念，为用户提供了一种简单的方式，以快速实现微服务的布置和管理。

下面从 3 个方面来阐述 Kubernetes 的核心功能（如图 5-4 所示）：自动化的任务管理、对微服务架构体系的支撑和最大化资源的利用率。

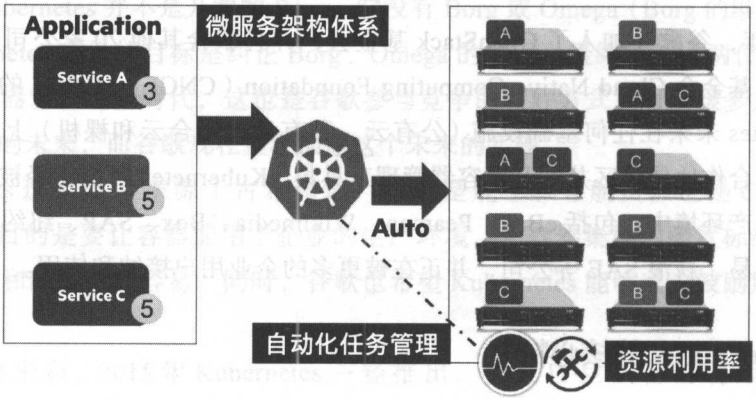


图 5-4 Kubernetes 的核心功能

1）Kubernetes 提供的第一组核心功能是自动化的任务管理，包括以下几方面的应用自动化管理功能，如图 5-5 所示。

- 应用自动发布功能。应用的发布是最常见的运维工作之一，在传统的 IT 环境中，运维人员通常需要进行大量的手工操作，包括数据库配置、负载均衡器配置、应用打包等工作才能完成。Kubernetes 则以一种智能的方式将应用程序发布到数据中心的各个服务器上去，力图减少大量的人力工作。
- 应用自动故障恢复功能，即实现应用的高可用。无论是主机发生故障，还是应用发生故障，Kubernetes 都能保证应用持续被监控，在发生故障时及时重启，以保证系统能够持续提供服务。在传统的 IT 环境中，通常还要搭建额外的监控系统，以及大量的脚本来实现应用的重启。
- 应用的扩容缩容功能。应用的可扩展性是现代分布式应用的一个基本要求。Kubernetes 提供了一种基于性能指标来动态地扩容或缩容应用实例数量的功能，以实现对业务波动的弹性支撑。

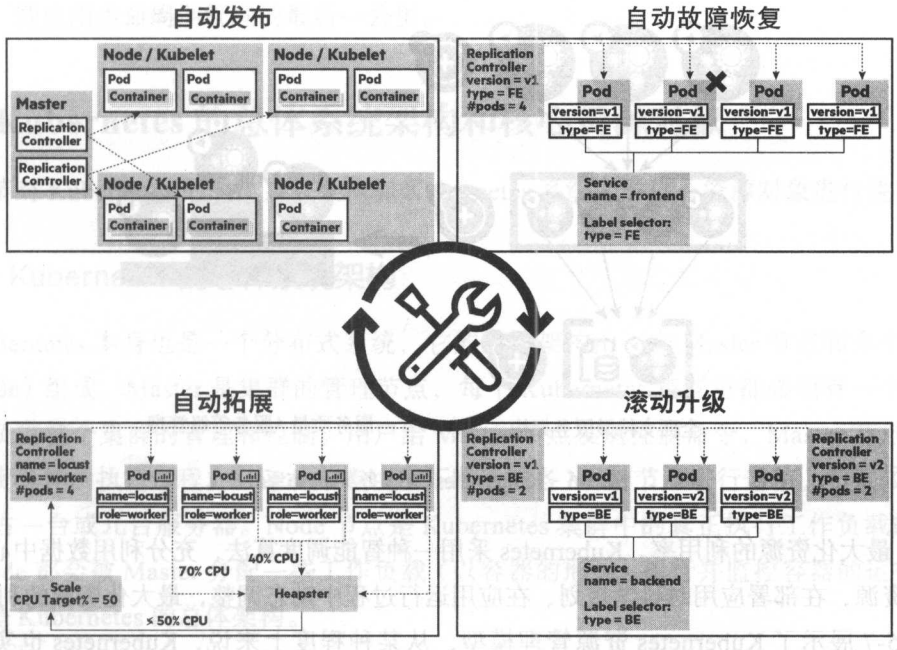


图 5-5 Kubernetes 的自动化任务管理功能

- 应用的滚动升级功能。在传统的 IT 系统运维工作中，应用的升级都是人工完成的，并常常伴随着业务的中断。Kubernetes 的滚动升级功能将应用逐个升级替换，逐渐完成整个应用集群的升级，以保证服务的不中断。

2）对微服务架构体系的支撑，如图 5-6 所示。在 Martin Fowler 的微服务理论体系中提到的“组件服务化”“智能端点与傻瓜管道”等特性，以及对微服务质量的精细化运营，正是 Kubernetes 提供的第二组核心功能。

- 组件服务化特性是指把单体应用系统中的组件拆分成多个微小的服务。智能端点指的是拥有独立完整业务逻辑的微服务，它们从管道上获取请求消息，进行智能处理，然后产生处理后的应答消息再放回管道。傻瓜管道指的是连接微服务进行消息传递的通信机制，它们只负责消息的传送，不关心业务逻辑的处理。Kubernetes 的微服务管理机制就是对这些微服务特性的完美支撑。
- Kubernetes 把服务的概念提升到机器之上，以支撑对微服务质量的精细化运营。对于用户来说，在使用 Kubernetes 的时候，完全不用关心底层的硬件、操作系统、系统资源等基础设施，而只要关注微服务本身的设计，以及微服务之间的调用关系，这样就能为不同的业务提供不同的精细化服务质量等级（SLA）。对于微服务上线之后的运维工作，Kubernetes 也希望运维人员将关注中心从应用如何发布转向业务级别的服务运维，关心业务运行指标，及时做出调整，为提供更精准的业务级别服务质量提供支持。

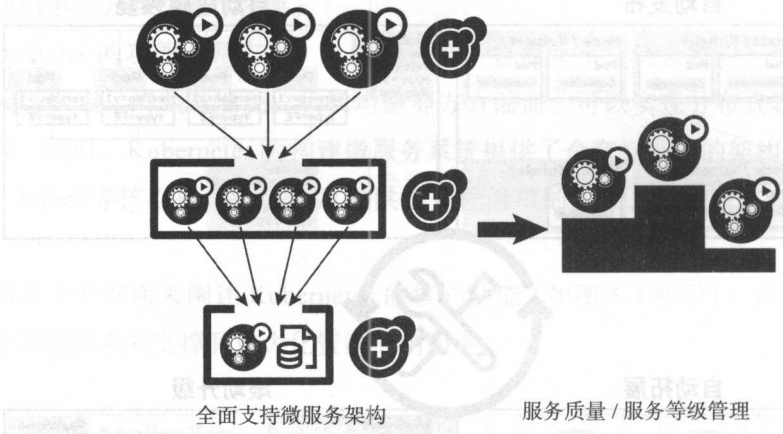


图 5-6 Kubernetes 微服务架构体系

3）最大化资源的利用率。Kubernetes 采用一种智能调度算法，充分利用数据中心里所有机器的资源，在部署应用时科学规划、在应用运行过程中动态调整，最大化资源的利用率。

图 5-7 展示了 Kubernetes 资源管理模型，从某种程度上来说，Kubernetes 也实现了一些 DC/OS（数据中心操作系统）的概念。从硬件的角度来说，Kubernetes 将所有机器当成一个整体资源池来管理，包括如何最大限度地提高机器的资源利用率，在哪些机器上创建哪些容器应用，什么时候增加，在哪些机器上增加，什么时候删除等工作。这些工作与传统 IaaS 平台的主要区别是 Kubernetes 管理的对象只是容器，而不是虚拟机。

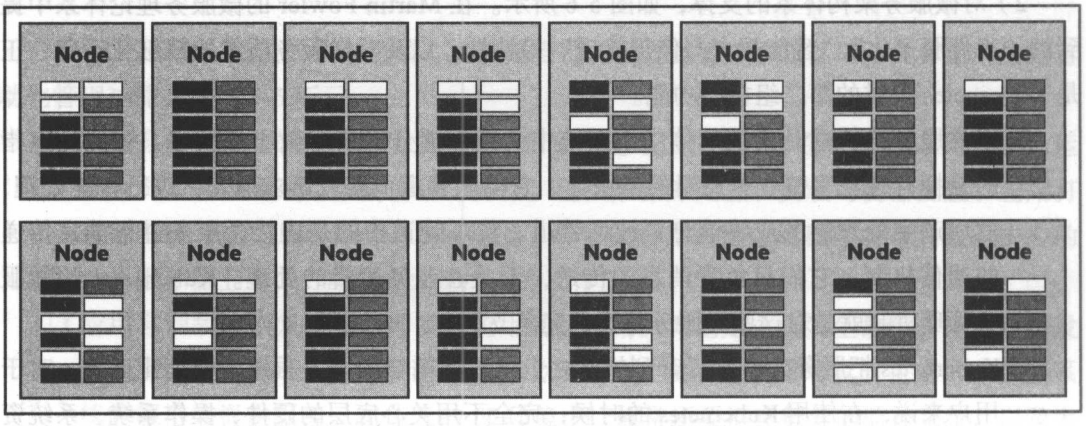


图 5-7 Kubernetes 资源管理模型

最后，简单阐述一下 Kubernetes 与 DevOps 的关系。DevOps 流水线包括从软件源代码的修改、提交、编译、打包到发布的全流程，Kubernetes 可以看作 DevOps 流水线的最后一

个环节，即应用生命周期管理的最后一公里。

5.2 Kubernetes 的总体系统架构和核心资源对象

本节对 Kubernetes 的总体系统架构和 Kubernetes 系统中的核心资源对象进行说明。

5.2.1 Kubernetes 的总体系统架构

Kubernetes 本身也是一个分布式系统，它的总体架构由一个 Master 节点和多个计算节点（Node）组成。Master 是集群的管理节点，每个 Kubernetes 集群里都必须有一个 Master 节点来负责整个集群的管理和控制。用户给 Master 节点发送控制命令，Master 节点接收指令后负责具体的执行过程。Master 节点由于不断地与各 Node 节点进行通信，负载较高，通常会独占一台或几台服务器。Node 节点是 Kubernetes 集群中的真正执行工作负载的节点，每个 Node 都会被 Master 分配一些工作负载（以容器的形式）执行并监控容器的运行状态。图 5-8 是 Kubernetes 的总体架构。

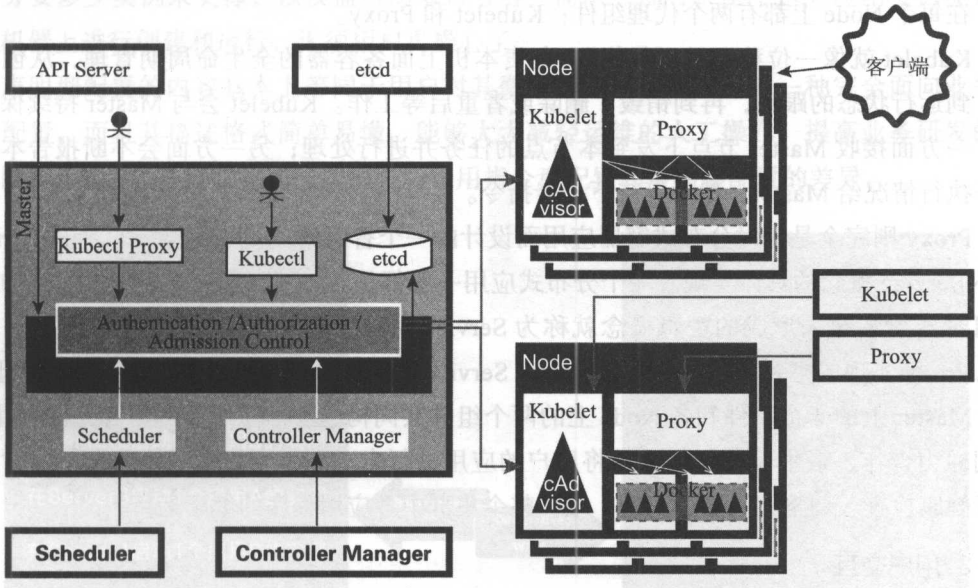


图 5-8 Kubernetes 的总体架构

下面对 Master 节点和 Node 节点的核心功能组件，以及用户如何与 Kubernetes 进行交互进行说明。

1. Master 节点

在 Master 节点里共有 4 个主要的功能组件，其中 API Server、Controller Manager 和

Scheduler 这 3 个组件是核心控制程序，作为集群的中央数据库 etcd，保存的是整个集群内所有容器应用的运行状态、任务配置信息和各种事件信息等内容。

API Server 提供了集群总管和总线的功能，为集群内各个功能模块之间提供数据交互和通信功能，并将各任务的信息保存到 etcd 中。所有需要与 Master 进行交互的其他功能组件都必须与 API Server 进行交互，不能直接去操作 etcd 数据库。

作为集群的管理控制中心，Controller Manager 负责集群内各种资源对象（包括 Node、Pod、Service、Endpoint、Namespace 等）的管理。当某个 Node 或容器发生故障时，Controller Manager 会及时执行自动化修复流程，确保容器应用始终处于用户预期的工作状态。

Scheduler 的主要功能是将待调度的 Pod 基于一定的调度算法和调度策略调度到集群中的某个合适的 Node 上。例如，基于各 Node 的状态给节点打分，经过一系列计算找到一个有可用资源（CPU、内存、磁盘空间）的合适节点，再根据用户设定的调度策略，最终为容器应用找到最适合的 Node 节点。然后通过 API Server 进一步通知 Controller Manager 控制中心把任务发到每个节点的代理上执行。

2.Node 节点

在每个 Node 上都有两个代理组件：Kubelet 和 Proxy。

Kubelet 就像一位称职的“保姆”，负责本机上面各容器的全生命周期管理，从创建开始，到运行状态的跟踪，再到销毁、删除或者重启等工作。Kubelet 会与 Master 持续保持通信，一方面接收 Master 节点下发到本节点的任务并进行处理；另一方面会不断报告本机的任务执行情况给 Master，然后等待下一步指令。

Proxy 则完全是针对分布式容器应用而设计的一个智能的负载均衡器，它是 Kubernetes 核心 Service 概念的具体实现。一个分布式应用一般都是由一组功能相同的容器组成的，将这组容器封装起来形成的逻辑概念就称为 Service（详见 5.2.2 节“3.Service”小节中的说明）。Proxy 实现的主要功能就是将客户端对 Service 的访问请求转发到后端正确的容器上去。

Master 上的 4 个组件和各 Node 上的两个组件共同构建起一个完整的 Kubernetes 集群，它们通力合作，最主要的工作就是将用户的应用进行自动的发布，然后连续不断地对应用进行健康检查、调整、重启等工作，维护整个集群中各应用程序的健康运行。

3. 用户交互

对于用户来说，应该如何与 Kubernetes 集群进行交互呢？这里有两种常见的交互模式。一种是集群中容器应用提供的服务，由于容器运行在各 Node 节点上，用户直接访问各 Node 就可以了。另一种是用户给集群发送的应用管理的控制指令，这就要跟 Master 节点打交道了。通过一些简单的配置和定义，用户可以使用命令行工具 Kubectl 或者 Master 提供的 API，将指令发送给 Master 节点（即 API Server），Master 节点就会处理用户的请求了。常见的请求包括创建容器、副本数量调整、删除容器等操作。

5.2.2 Kubernetes 的核心资源对象

Kubernetes 在对集群进行管理时，采用了称为资源对象（Resource Object）的统一模型来定义各种各样的被管理对象，既用于用户提交工作任务请求，也用于系统内部各模块之间的交互。

资源对象是 Kubernetes 系统中最核心的管理单元，包括 Node、Pod、RC、Service、Endpoint、Namespace 等，所有的资源对象的定义和运行状态都保存在 etcd 数据库中，并可以进行增、删、改、查等操作。Kubernetes 通过持续跟踪并对比 etcd 数据库中保存的“用户期望状态”与“实际运行状态”的差异，来实现自动控制和自动纠错等高级功能。

为了描述用户期望的系统运行状态，Kubernetes 创新性地使用一种“声明型”的配置供用户进行描述。相对而言，在传统的工厂运维工作中，运维工程师使用的多是“指令型”的配置方式，并常伴随着各种自动化脚本。指令型配置的核心是运维人员必须“知道”在哪些机器上启动或停止哪些应用程序，与机器深度绑定。而在声明型的配置中，Kubernetes 希望用户不用再关心机器，而只要关心应用本身。换句话说，Kubernetes 只需用户给出一个服务要多少实例来支撑，以及监听的端口号等“需求”即可，至于这些应用实例应该在哪台机器上进行创建和运行，无须用户再操心了。

声明型配置的内容基本上等同于用户对其微服务的运行需求，是一种完全面向业务需求的配置，而且其描述格式简单易懂，能够大大减轻运维的人工操作，提高业务研发的效率。图 5-9 描述了一种常见的应用部署场景用指令型配置和声明型配置的差异。

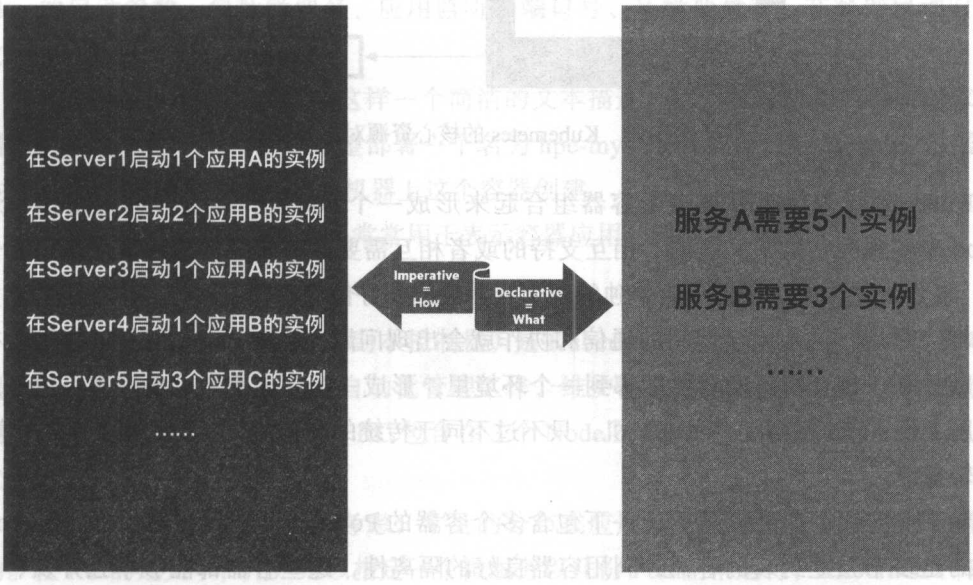


图 5-9 指令型配置与声明型配置

声明型配置可以用 YAML 语言或 JSON 语言进行描述，YAML 是更加易读的文本格式，而 JSON 多用于程序化使用。在本章后续的示例中，将使用 YAML 语言格式对资源对象进行定义和说明。

下面对 Kubernetes 的 4 个最核心的资源对象进行详细说明，包括 Pod、RC、Service 和 Label/Label Selector。

1. Pod

Pod 是谷歌独创的一个概念，从 Borg 系统借鉴而来。在 Kubernetes 系统中，最小的应用管理单元是 Pod，而不是容器。与单个容器最大的区别是：一个 Pod 可以包括多个容器。同一个 Pod 内的容器拥有相同的 IP 地址，还可以共享存储卷（Volume）和网络栈（Network Stack）。一个 Pod 内的多个容器共享一个 IP 地址，不同的容器使用不同的端口号提供服务，这样一个 Pod 能提供多个端口的服务。图 5-10 描述了 Pod 的架构和特性。

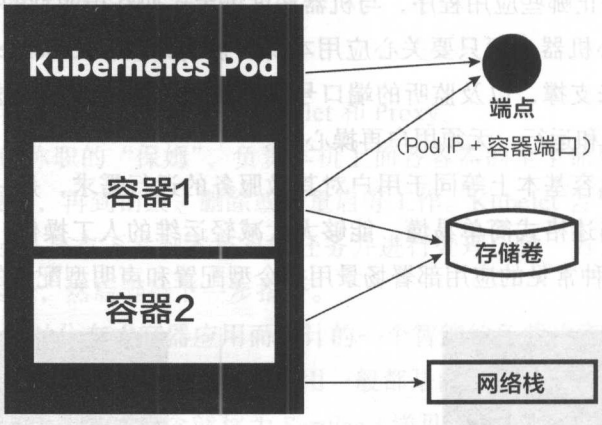


图 5-10 Kubernetes 的核心资源对象 Pod

Kubernetes 为什么要把多个容器组合起来形成一个 Pod 呢？什么样的应用比较适合用 Pod 来实现呢？紧密关联的、相互支持的或者相互需要共享资源的应用程序都适合使用 Pod。将紧密关联的应用拆分为单独的两个容器看起来好像没有什么问题，但如果容器实例数量增大之后，这两组容器间的通信和协作就会出现问题。Kubernetes 的 Pod 模型把很多个紧密关联、相互支持的容器捆绑到一个环境里，形成一个多应用组合的完整系统。从某种角度上说，Pod 相当于一台虚拟机，只不过不同于传统的虚拟机，Pod 使用容器技术实现，更加轻量级。

对于多应用的组合，再对比一下包含多个容器的 Pod 和包含多个进程的容器。Pod 内包含的是多个独立封装的容器，利用容器良好的隔离性，这些容器都能够单独开发、单独更新。而对于包含多个进程的一个容器来说，就显得比较臃肿了。当其中一个应用程序代

码需要修改时，整个镜像就必须重新打包。所以 Pod 是更加灵活的对容器的二次封装。多容器组合的 Pod 在 Kubernetes 的基础系统服务组件中很常见，许多 Pod 都是由四五个容器组合而成的。这也是 Borg 系统的最佳实践之一。

下面用 YAML 格式的配置来说明一个 Pod 的定义。（YAML 的语法类似于 PHP，对于空格的个数有严格的要求，详见 http://yaml.org。）

```
apiVersion: v1
kind: Pod
metadata:
  name: myweb
  labels:
    app: myweb
spec:
  containers:
  - name: hpe-myweb
    image: hpe-myweb:v1
    ports:
    - containerPort: 8080
    env:
    - name: MYSQL_SERVICE_HOST
      value: 'hpe-mysql'
    - name: MYSQL_SERVICE_PORT
      value: '3306'
```

Pod 的定义简单明了，apiVersion 固定为“v1”，kind 固定为“Pod”，然后在 metadata 部分为 Pod 设置名称。之后的 spec 部分完全是容器的参数定义，基本上一一对应到 Docker 的启动参数，包括镜像名、应用监听的端口号、环境变量等，在容器启动时加载生效。

在 Kubernetes 平台上，仅用这样一个简洁的文本描述，就能够对应用的部署需求进行说明了。在这个例子中，用户希望部署一个名为 hpe-myweb 的容器，Kubernetes 根据这份“需求”就能自动地在一台合适的机器上这个容器创建。

在本章后文的描述中，Pod 也常常用于表示容器应用，请读者自行甄别。

2. RC

RC（Replication Controller，副本控制器）是 Kubernetes 系统的另一个核心资源对象。它提供对一个或多个 Pod 副本的自动化管理工作，维护任意时刻运行的 Pod 副本数量都符合用户的期望。图 5-11 描述一个 RC 包含在 3 个 Node 的集群中始终维护着 4 个 Pod 副本的状态。

出于性能和高可用性方面的考虑，常见的分布式应用都是通过启动多份实例来提供服务的。在 RC 的定义中，用户可以指定一个微服务以几份 Pod 副本运行，Kubernetes 将保证实际运行的 Pod 数量总是与用户期望的副本数量相等。当某台 Node 宕机或者 Pod 应用崩

溃时，Kubernetes 会自动创建新的 Pod 来替换无效的 Pod。

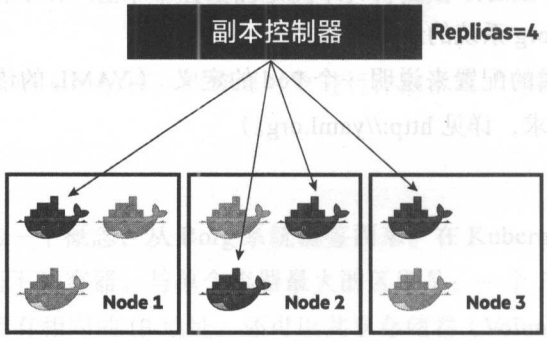


图 5-11 Kubernetes 的核心资源对象 RC

Kubernetes 基于 RC 实现了容器应用的自动化管理功能，从确保用户期望的 Pod 数量、故障恢复、扩容缩容到滚动升级，基本上实现了应用级别的高可用。在 Master 节点里 Controller Manager 模块中也有同名的 Replication Controller 子模块，所以 RC 既可以表示 Master 节点的控制器模块，也可以表示资源对象。

下例描述了一个 RC 的定义。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hpe-myweb
spec:
  replicas: 3
  selector:
    app: hpe-myweb
  template:
    metadata:
      labels:
        app: hpe-myweb
spec:
  containers:
  - name: hpe-myweb
    image: hpe-myweb:v1
    ports:
    - containerPort: 8080
    env:
    - name: MYSQL_SERVICE_HOST
    value: 'mysql'
    - name: MYSQL_SERVICE_PORT
    value: '3306'
```

在 RC 的定义中，kind 为 ReplicationController。在 spec 部分中，与 Pod 的定义不同，

RC 的定义有两个新的内容 replicas 和 selector。replicas 定义的是用户期望运行的 Pod 副本数量，也是首次创建时的副本数量。在初次创建和运行之后，用户随时都可以调整 Pod 的副本数量。selector 将在后面进行详细说明。从 template 部分开始就都是 Pod 的定义了。

3. Service

在 Kubernetes 中，Service 是最核心的资源对象之一，Kubernetes 里的每个 Service 其实就是微服务架构中的一个个“微服务”，而 Pod、RC 等资源对象其实都是为实现 Service 而提供底层的基础支持的。

在微服务架构里，一个微服务通常都是由应用的多个实例组成的，并且支持水平扩展的分布式服务。在 Kubernetes 系统中，应用的实例由 Pod 实现，应用的高可用性由 RC 提供。但是由于 Pod 以轻量级的容器作为载体，容器的状态可能发生动态变化，如切换机器、扩容缩容等，因此对于客户端应用来说，动态地发现所有提供服务的 Pod 地址变得非常困难。Service 提供的主要功能就是为客户端访问后端服务提供入口，并将客户端请求转发到多个 Pod 的代理服务器，解决微服务架构中核心的服务发现、服务路由、负载均衡等问题。

类似于呼叫中心为客户提供的唯一客服电话号码，用户无须知道该号码后面具体由哪个客服人员提供服务，Service 也为客户端应用提供唯一稳定的入口地址，屏蔽后端的多个 Pod，以及 Pod 的变化。所以，Service 提供的是一种“智能”的负载均衡器的功能。这样，系统管理员就无须手工设置额外的负载均衡器来将客户端请求转发到各个应用实例上了，如图 5-12 所示。

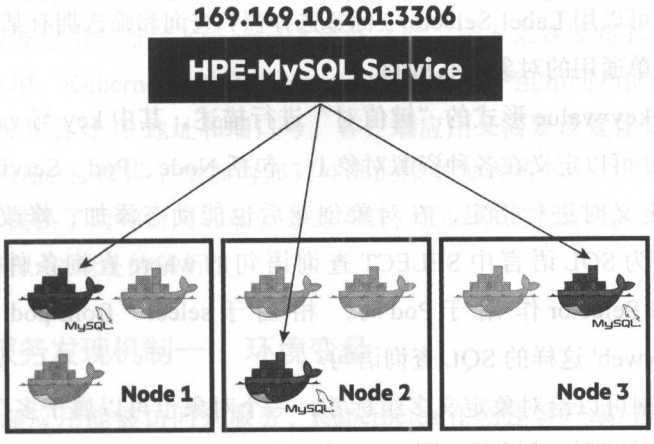


图 5-12 Kubernetes 的核心资源对象 Service

Service 提供的对客户端的访问入口是标准的 TCP/IP 网络地址，以 IP 地址和端口号来表示，并且支持不同的负载分发策略，包括轮询（Round Robin）和会话粘连（Session

Affinity)。

一个 Service 的定义如下。

```
apiVersion: v1
kind: Service
metadata:
  name: hpe-mysql
spec:
  ports:
  - port: 3306
  selector:
    app: hpe-mysql
```

在 Service 的定义中可以看到，由于 Service 是在 Pod 之上的更高层抽象，所以仅需定义客户端要访问的端口号，以及与 Pod 关联的 Selector。Service 的 IP 地址一般由 Kubernetes 的 Master 进行分配，用户也可以自行指定。关于 Service 的 IP 地址以及服务的发现机制，将在下一节进行详细的说明。

4. Label/Label Selector

细心的读者在 Replication Controller 和 Service 的定义中会发现一个名为 Selector 的字段。这就是管理对象与被管理对象之间的关联性设置了。在 Kubernetes 系统中，引入 Label 和 Label Selector 的定义来实现这种关联性，这也是 Borg 系统的另一个最佳实践。

Label 相当于我们熟悉的“标签”，给某个资源对象定义一个 Label，就相当于给它贴了一个标签，然后就可以用 Label Selector（标签选择器）查询和筛选拥有某些 Label 的资源对象，实现了一种简单通用的对象关联机制。

一个 Label 以 key=value 形式的“键值对”进行描述，其中 key 与 value 由用户指定为任意字符串。Label 可以定义在各种资源对象上，包括 Node、Pod、Service、RC 等。Label 通常在资源对象定义时进行指定，在对象创建后也能动态添加、修改或者删除。Label Selector 可以类比为 SQL 语言中 SELECT 查询语句的 where 查询条件，例如，app:hpe-myweb 这个 Label Selector 作用于 Pod 时，相当于 select * from pod where pod's labels contain 'app:hpe-myweb' 这样的 SQL 查询语句。

使用 Label 机制可以给对象定义多组标签，一个对象也可以属于多个组，这样就比给对象定义静态的属性更加灵活了。图 5-13 描述了一个 Service 和两个 RC 用不同的 Label Selector 关联到不同的 Pod。

在 Kubernetes 系统中，运行和管理的最小单元是 Pod，结合管理工具 RC 和 Service（用 Label Selector 关联 Pod）就可以实现一个自动化微服务运行管理平台了。

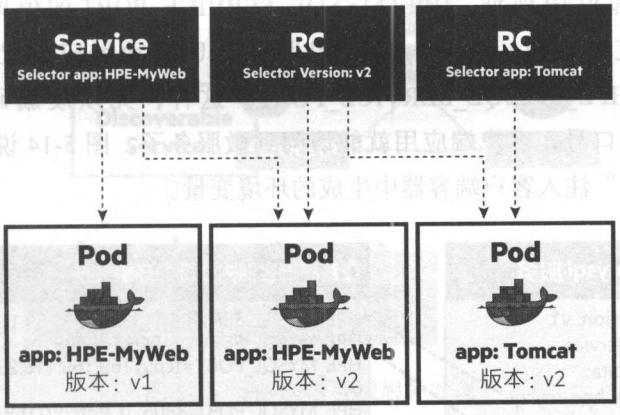


图 5-13 Kubernetes 的核心资源对象 Label 和 Label Selector

5.3 Kubernetes 的服务发现机制

服务发现机制是微服务架构里最重要的功能，也是微服务支撑平台必须实现的功能。如果服务发现没有实现，客户端应用将很难找到提供服务的应用访问地址，更困难的是，当后端服务运行过程地址发生了变化时，客户端应该如何重新访问。服务发现要求定义好应用之间的调用方式、通信协议、负载均衡方式等机制，通常要用多个组件才能搭建起一个完整的服务发现机制，而在 Kubernetes 系统中服务发现机制是内置开箱即用的功能。

在用户的微服务应用发布到 Kubernetes 集群之后，接下来就要将其访问地址以某种形式提供给客户端应用。Kubernetes 抽象出 Service 的概念为一组相同功能的 Pod 提供统一的入口，并为 Service 设置好 IP 地址和端口号。客户端应用又需要设置什么 URL 地址才能访问到它们呢？ Kubernetes 提供了集群内部和外部的服务发现机制。对于集群内部的客户端应用，Kubernetes 提供了两种访问后端服务的方式：①以环境变量形式；②以虚拟 DNS 服务实现服务名到虚拟 IP 地址的解析。

5.3.1 集群内服务发现机制一：环境变量

为了让客户端应用能够访问到服务，Kubernetes 在创建客户端应用时，直接将服务的信息以环境变量的方式注入容器内部环境（env）中，这样客户端应用程序就可以根据一些简单的规则来配置服务的地址了。例如，一个微服务名为 hpe-mysql，在客户端容器中，Kubernetes 将为该服务设置一系列环境变量，其中最重要的有两个：HPE_MYSQL_SERVICE_HOST 和 HPE_MYSQL_SERVICE_PORT。HPE_MYSQL_SERVICE_HOST 的

值即为 Service 的虚拟 IP 地址，HPE_MYSQL_SERVICE_PORT 的值即为 Service 的端口号。对于客户端应用来说，访问 hpe-mysql 服务的 URL 地址应配置为 $HPE_MYSQL_SERVICE_HOST:$HPE_MYSQL_SERVICE_PORT。这样，无须硬编码（hardcode）服务的虚拟 IP 地址和端口号，客户端应用就能访问到微服务了。图 5-14 说明了 Kubernetes 将 Service“hpe-mysql”注入客户端容器中生成的环境变量。

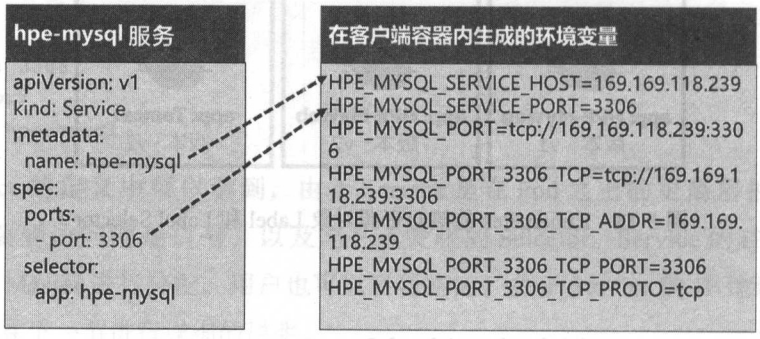


图 5-14　Service 以环境变量方式注入客户端容器中

使用环境变量方式简单清晰，客户端应用的配置也很简单，但有一个明显的问题：被访问服务必须先于客户端应用创建出来。如果客户端应用在服务定义之前就创建完成，然后再定义服务，则对于一个已经启动完成的容器应用来说，再把服务的定义以环境变量的方式注入进去，也就没有什么意义了。

由于环境变量方式的局限性，Kubernetes 引入虚拟 DNS 服务，把微服务的名称设置为 DNS 域名，使客户端应用能够用服务名来访问微服务。下面对这种机制进行说明。

5.3.2　集群内服务发现机制二：DNS 服务

第二种服务发现机制使用一个虚拟 DNS 服务（SkyDNS）来实现。为了让客户端应用能够以固定的地址访问微服务，客户端应用只要知道服务的名称，不用关心服务的 IP 地址，也无须使用环境变量。SkyDNS 将服务名与服务 IP 地址的对应关系记录为 DNS 记录，并提供客户端访问服务名时的 IP 地址解析功能。SkyDNS 也会对 DNS 记录进行维护，包括创建、修改和删除等操作。

这个 SkyDNS 服务将作为 Kubernetes 集群中的 DNS 服务器，其地址将设置到所有容器环境中的 /etc/resolv.conf 文件中，应用程序就可以像访问普通域名一样访问 Service 了。仍以上一节的 hpe-mysql 服务为例，在客户端应用容器中，访问“hpe-mysql:3306”这个地址就能访问到 hpe-mysql 服务了，如图 5-15 所示。

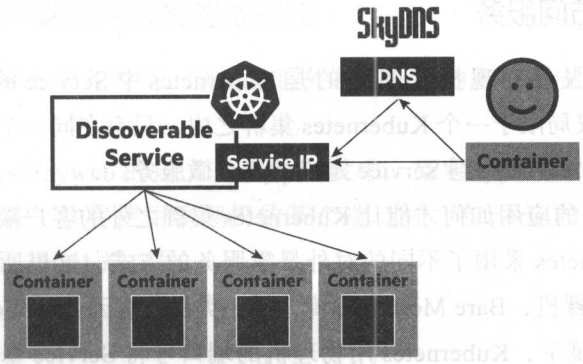


图 5-15 SkyDNS 完成服务发现

我们知道 Service 的表现形式是 IP 地址和端口号，在网络分层模型中的 TCP/IP 层（4 层）。这种机制能够满足大多数客户端到服务端的网络请求转发。不过，对于 HTTP 服务来说，不同的域名或 URL 路径经常对应不同的后端服务或者虚拟服务器（virtual host），即 HTTP 层（7 层）的路由转发。Kubernetes 系统引入 Ingress 机制将不同 URL 的访问请求转发到后端不同的 Service，以实现 7 层路由的功能。Ingress 包括创建一个 Ingress 资源对象（路由规则的设置）和启动一个 Ingress Controller（转发请求的具体实现）。

图 5-16 显示了一个使用 Ingress 实现典型 7 层路由的例子。

- 对 http://mywebsite.com/api 的访问将被路由到后端名为“api”的 Service。
- 对 http://mywebsite.com/web 的访问将被路由到后端名为“web”的 Service。
- 对 http://mywebsite.com/docs 的访问将被路由到后端名为“docs”的 Service。

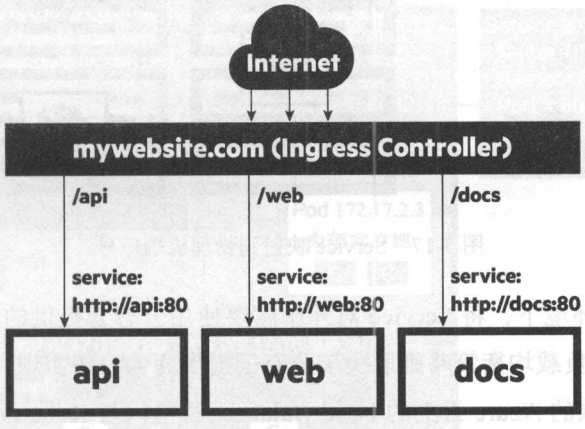


图 5-16 Ingress 示例

5.3.3 从集群外访问服务

前两节讲述的服务发现机制使用的是Kubernetes中Service的虚拟IP地址，所以Service的作用范围仅局限于一个Kubernetes集群之内，只有在同一个Kubernetes集群内的其他客户端容器应用才能访问到Service定义的各个微服务。

但对外提供服务的应用如何才能让Kubernetes集群之外的客户端应用访问到呢？在不同的环境下，Kubernetes采用了不同的对外暴露服务的方式。这里所说的环境主要包括两类：一类是物理机（裸机，Bare Metal）环境，另一类是公有云（Public Cloud）环境。

1）在物理机环境下，Kubernetes用物理机的端口号将Service暴露出去，使客户端应用可以通过物理机的IP地址和端口号来访问Service。需要说明的是，Service端口号映射到物理机的某个端口号与Docker容器应用容器端口号映射到物理机是不同的，Service映射到物理机的后端仍然是虚拟Service的访问地址，而不是容器，Service还要完成将请求负载分发到各个容器的过程。具体的设置方式也很简单，只需修改Service的定义，指定type=NodePort即可。物理机具体使用哪个端口号，用户可以指定，也可以交给Kubernetes系统自动分配。图5-17表示某个Service使用物理机的30001端口号对外提供服务，客户端应用访问集群中任意一台Node的30001端口号都能够访问到Service了。

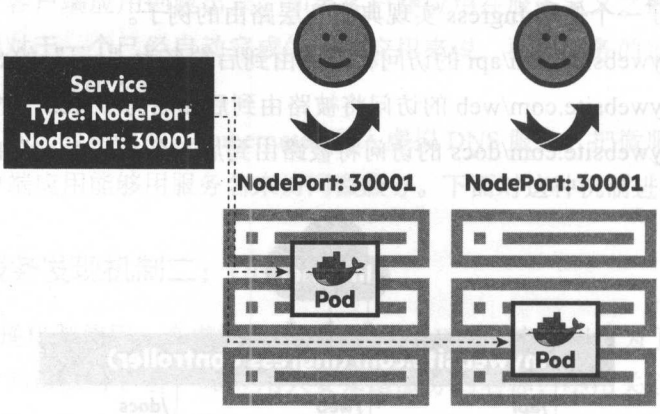


图5-17 Service映射到物理机端口号

2）而在公有云环境下，将Service对外暴露要使用公有云提供的Load Balancer，Load Balancer能够提供的负载均衡策略则取决于公有云提供商各自的实现。目前，谷歌的GCE、亚马逊的AWS、微软的Azure云上的Load Balancer都可以与Kubernetes的Service进行对接，为外部客户端提供微服务的访问地址。

5.3.4 集群内外客户端访问服务的数据流

本节将以 hpe-myweb 服务为例，说明集群外部和内部的客户端应用在访问该微服务时的数据流。

下面是 Service hpe-myweb 的定义。在 spec 段中，指定 type=NodePort 表示该 Service 将对集群之外提供服务，并将使用物理机的某个端口号。在 ports 段中的 nodePort=30001 则指定了使用物理机的 30001 端口号。

```
apiVersion: v1
kind: Service
metadata:
  name: hpe-myweb
spec:
  type: NodePort
  ports:
    - port: 8080
      nodePort: 30001
  selector:
    app: hpe-myweb
```

图 5-18 显示了从集群外部或内部访问 hpe-myweb 服务，请求是如何经过 Kubernetes 系统转发到正确的容器的数据流。

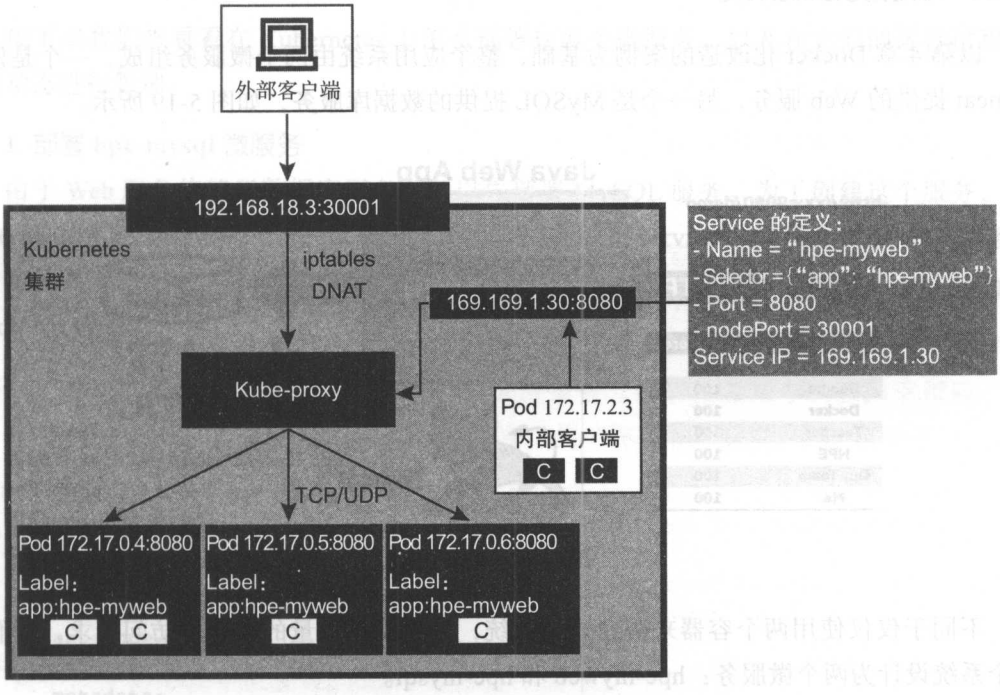


图 5-18 集群内外部访问 hpe-myweb 服务的数据流

假设物理机的IP地址是192.168.18.3，外部客户端访问hpe-myweb时的URL地址为192.168.18.3:30001。当该物理机接收到这个请求之后，kube-proxy根据Service的定义查询出要转发到后端Pod的列表，本例中有3个Pod提供真正的服务，它们的容器IP地址分别是172.17.0.4、172.17.0.5、172.17.0.6。同时，系统还会完成端口号的映射。物理机的30001端口号对应的是Service的8080端口号，再负载分发到各Pod的8080端口号上。对物理机192.168.18.3:30001的访问最终会被转发到172.17.0.4:8080、172.17.0.5:8080172.17.0.6:8080三者之一上。

对于集群内部的客户端来说，它只需访问Service本身的地址，如果有DNS服务，则访问URL为“hpe-myweb:8080”，当然，如果明确知道Service的虚拟IP地址，访问URL也可以是“169.169.1.30:8080”。之后的数据转发就与外部客户端的访问一样了。

5.4 一个完整Kubernetes的微服务案例

本节将通过一个案例，对如何在Kubernetes平台上实现基于微服务架构的应用部署过程进行详细说明。

5.4.1 微服务系统架构

以第4章Docker化改造的案例为基础，整个应用系统由两个微服务组成，一个是使用Tomcat提供的Web服务，另一个是MySQL提供的数据库服务，如图5-19所示。

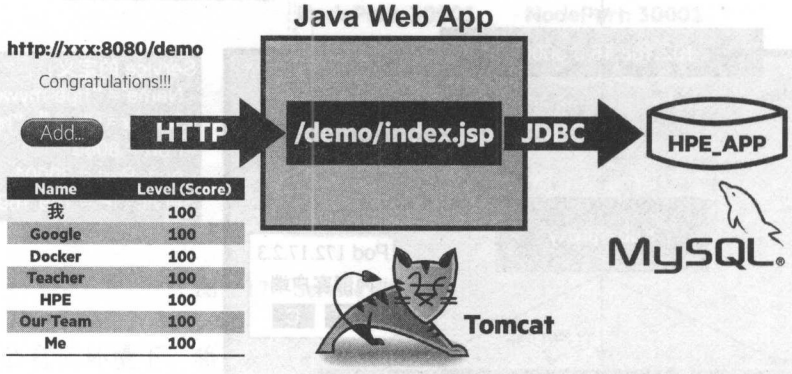


图5-19 系统架构

不同于仅仅使用两个容器来搭建这个系统，为了支撑大量的客户端访问请求，我们将整个系统设计为两个微服务：hpe-myweb和hpe-mysql。

hpe-myweb服务是一个Web服务，初始运行5个实例；hpe-mysql服务是MySQL数据

库服务，初始运行 3 个实例，如图 5-20 所示。

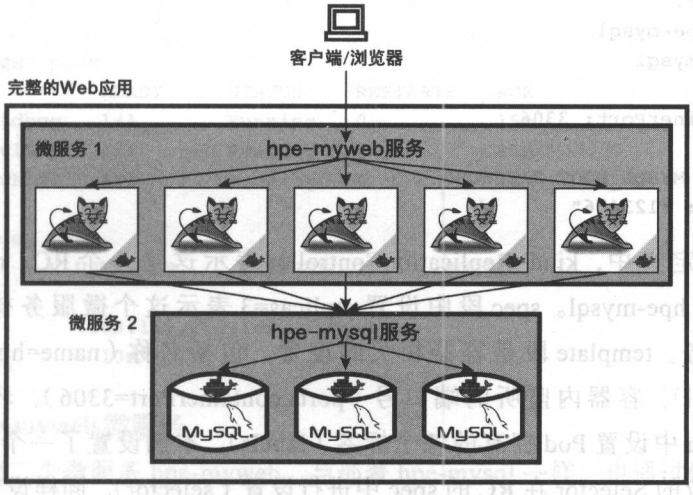


图 5-20 微服务系统架构

5.4.2 在 Kubernetes 上部署微服务

接下来我们来看看在 Kubernetes 上怎么部署这两个微服务，以及在它们的运行过程中如何动态进行管理。

1. 部署 hpe-mysql 微服务

由于 Web 服务依赖于数据库服务，所以先部署 MySQL 服务。为了创建这个服务，在 Kubernetes 上只要创建一个 Replication Controller 和一个 Service 总共两个资源对象就能完成。编辑两个 YAML 格式的“声明型”配置文件来定义它们：hpe-mysql-rc.yaml 和 hpe-mysql-svc.yaml。

RC 定义文件 hpe-mysql-rc.yaml 的内容如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hpe-mysql
spec:
  replicas: 3
  selector:
    app: hpe-mysql
  template:
    metadata:
      labels:
```

```
    app: hpe-mysql
spec:
  containers:
  - name: hpe-mysql
    image: mysql
    ports:
    - containerPort: 3306
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "123456"
```

在这个RC定义中，kind=ReplicationController表示这是一个RC。metadata段中定义RC的名称为hpe-mysql。spec段中设置replicas=3表示这个微服务初始将启动3个实例来提供服务。template段是容器相关的设置，包括名称（name=hpe-mysql）、镜像名（image=mysql）、容器内监听的端口号（ports.containerPort=3306）、环境变量等。在template.metadata中设置Pod拥有的各个标签（labels），本例设置了一个标签：app=hpe-mysql。与之对应的Selector在RC的spec中进行设置（selector），同样设置的是app=hpe-mysql，这样RC就能够与拥有该标签的Pod关联起来了。

Service定义文件hpe-mysql-svc.yaml的内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: hpe-mysql
spec:
  ports:
  - port: 3306
  selector:
    app: hpe-mysql
```

其中，metadata.name设置Service的服务名。spec段中的ports定义Service的端口号，这里设置为3306，与RC中容器监听的端口号一致。在spec中设置selector，与RC一样，设置的是app=hpe-mysql，这样Service就能够与拥有该标签的Pod关联起来了。

编辑完成这两个配置文件后，登录到Kubernetes集群的Master节点服务器上，通过Kubectl命令行工具将这两个“声明”提交给Master。

```
# kubectl create -f hpe-mysql-rc.yaml
replicationcontroller "hpe-mysql" created

# kubectl create -f hpe-mysql-svc.yaml
service "hpe-mysql" created
```

这两个步骤执行完毕后，第一个微服务hpe-mysql就部署完成了。用Kubectl工具检查一下容器是否运行正常：

```
# kubectl get rc
NAME         DESIRED   CURRENT   READY     AGE
hpe-mysql    3         3         3         10s

# kubectl get pods
NAME               READY     STATUS    RESTARTS   AGE
hpe-mysql-tdbqg    1/1       Running   0          15s
hpe-mysql-v107p    1/1       Running   0          15s
hpe-mysql-vhbd3    1/1       Running   0          15s

# kubectl get svc
NAME          CLUSTER-IP      EXTERNAL-IP     PORT(S)
AGE
hpe-mysql          169.169.253.143   <none>
3306/TCP           10s
```

2. 部署 hpe-myweb 微服务

然后部署第二个微服务 hpe-myweb。与部署 hpe-mysql 一样，也通过创建一个 RC 和一个 Service 来完成。

RC 的定义文件 hpe-myweb-rc.yaml 的内容如下：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hpe-myweb
spec:
  replicas: 5
  selector:
    app: hpe-myweb
  template:
    metadata:
      labels:
        app: hpe-myweb
    spec:
      containers:
      - name: hpe-myweb
        image: kubeguide/tomcat-app:v1
        ports:
        - containerPort: 8080
        env:
        - name: MYSQL_SERVICE_HOST
          value: 'hpe-mysql'
        - name: MYSQL_SERVICE_PORT
          value: '3306'
```

其中，在 spec 段中设置 replicas=5 表示这个微服务初始将启动 5 个实例来提供服务。template 段是容器相关的设置，包括名称（name=hpe-myweb）、镜像名（image=kubeguide/

tomcat-app:v1）、容器内监听的端口号（ports.containerPort=8080）。环境变量MYSQL_SERVICE_HOST=hpe-mysql是该应用最重要的设置，因为在其容器启动时，会用这个环境变量的值建立与hpe-mysql服务的数据库连接。在本例中配置的是Service hpe-mysql的名称，要求在Kubernetes集群中DNS服务正常运行（参见5.3.2节）。为了简便，将另一个环境变量MYSQL_SERVICE_PORT的值直接设置为3306。

Service的定义文件hpe-myweb-svc.yaml的内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: hpe-myweb
spec:
  type: NodePort
  ports:
    - port: 8080
      nodePort: 30001
  selector:
    app: hpe-myweb
```

设置type=NodePort和nodePort=30001表示将这个服务映射到物理机的30001端口上，以供集群外部的客户端进行访问。

登录到Master节点的服务器上，再次用Kubectl工具将这两份“声明”提交给Master。

```
# kubectl create -f hpe-myweb-rc.yaml
replicationcontroller "hpe-myweb" created

# kubectl create -f hpe-myweb-svc.yaml
service "hpe-myweb" created
```

执行完成后，第二个微服务hpe-myweb也就部署完成了。用kubectl get命令查看RC、Service的状态和Pod的运行状态：

```
# kubectl get rc
NAME         DESIRED   CURRENT   READY     AGE
hpe-mysql    3         3         3         1m
hpe-myweb    5         5         5         10s

# kubectl get pods
NAME               READY     STATUS    RESTARTS   AGE
hpe-mysql-tdbqg    1/1       Running   0          1m
hpe-mysql-v107p    1/1       Running   0          1m
hpe-mysql-vhbd3    1/1       Running   0          1m
hpe-myweb-h36m8    1/1       Running   0          10s
hpe-myweb-mhfvp    1/1       Running   0          10s
hpe-myweb-s8tzb    1/1       Running   0          10s
hpe-myweb-0f254    1/1       Running   0          10s
```

```
hpe-myweb-wvb8h   1/1       Running   0          10s

# kubectl get svc
NAME          CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
hpe-mysql                  169.169.253.143      <none>
3306/TCP            1m
hpe-myweb                  169.169.149.215      <nodes>
8080:30001/TCP        10s
```

到此两个微服务就都部署完成了。其中微服务 hpe-mysql 运行了 3 个应用实例，微服务 hpe-myweb 运行了 5 个实例，在 Kubernetes 集群中总共运行了 8 个实例。基于“声明型”配置，在整个部署的过程中，我们没有（也无须）指定到哪些具体的机器上去运行这 8 个实例，是不是非常简单？读者可以对比一下，在传统 IT 运维时代，需要哪些手工工作才能完成这 8 个应用实例的部署。

3. 在浏览器中访问微服务 hpe-myweb

在 Kubernetes 集群中，任意选一个 Node 节点，如 192.168.18.3，打开一个浏览器窗口，在地址栏输入 http://192.168.18.3:30001/demo/，将会看到 hpe-myweb 服务提供的页面，如图 5-21 所示。

Congratulations!!

Add...

Name	Level(Score)
google	100
docker	100
teacher	100
HPE	100
our team	100
me	100

图 5-21 浏览器访问 hpe-myweb 网页

页面中表格的数据是 hpe-myweb 进一步访问后端的 hpe-mysql 服务，从 MySQL 数据库中查询出来的数据。对于用户来说，其实并不知道后端有几个服务，以及每个服务由几个实例来支撑。

5.4.3 Kubernetes 自动化管理微服务示例

本节通过几个应用程序运行过程中的常见场景，对 Kubernetes 强大的自动化功能进行说明。

1. 自动应用故障恢复

在系统运行的过程中，有很多情况都可能导致应用程序无法提供服务，常见的异常情况包括：主机宕机（应用自然也就不存在了）；应用自身存在缺陷，运行一段时间后僵死；人为误删除等。Kubernetes 对应用的运行状态进行持续不断的监控，在异常情况发生时及时创建新的应用实例来恢复服务的可用性，这也被称为“自愈”(self-healing) 的系统。整个过程自动完成，无须人为干预。

例如，我们用 kubectl delete 命令删除一个正在运行中的 Pod 来模拟一种异常场景：

```
# kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
hpe-myweb-h36m8   1/1       Running   0          41s
hpe-myweb-mhfvp   1/1       Running   0          41s
hpe-myweb-s8tzb   1/1       Running   0          41s
hpe-myweb-0f254   1/1       Running   0          41s
hpe-myweb-wvb8h   1/1       Running   0          41s

# kubectl delete pod hpe-myweb-h36m8
pod "hpe-myweb-h36m8" deleted
```

再次查看运行中的 Pod 列表，可以看到一个新的 Pod（名为“ hpe-myweb-7btqg”）已经被创建出来并正确运行了，保证该微服务始终维持有 5 个副本提供服务。

```
# kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
hpe-myweb-7btqg   1/1       Running   0          20s
hpe-myweb-mhfvp   1/1       Running   0          2m
hpe-myweb-s8tzb   1/1       Running   0          2m
hpe-myweb-0f254   1/1       Running   0          2m
hpe-myweb-wvb8h   1/1       Running   0          2m
```

2. Pod 实例数量扩缩容

当某个微服务的 Pod 实例数量无法满足对当前业务请求量的处理时，Kubernetes 可以对其进行扩容，以提供更大的服务能力；反之，如果 Pod 实例的数量太多，也可以减少 Pod 实例的数量，以释放出更多可用资源。

以微服务 hpe-myweb 为例，其最初由 3 个 Pod 副本提供服务。当客户端请求增加时，需要水平扩展到 5 个 Pod 副本；而当客户端请求数量降低时，1 个 Pod 副本就足以提供服务。Pod 副本数量的变化对客户端是透明的，对客户端来说 Service 总是可用的，如图 5-22 所示。

使用 kubectl scale 命令对运行中的 hpe-myweb 服务进行扩容和缩容操作，以实现这个微服务的水平扩展。扩缩容的执行过程如下。

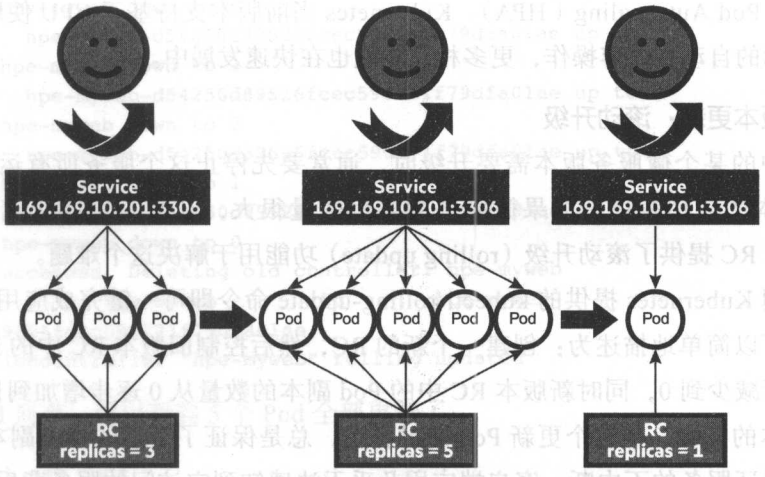


图 5-22 容器应用的扩缩容

初始时有 3 个 Pod 副本：

```
# kubectl get pods
hpe-myweb-78t6p    1/1       Running    0          4s
hpe-myweb-hs0gr    1/1       Running    0          4s
hpe-myweb-jqbjt    1/1       Running    0          4s
```

扩容为 5 个 Pod 副本：

```
# kubectl scale rc hpe-myweb --replicas=5
replicationcontroller "hpe-myweb" scaled

# kubectl get pods
NAME               READY     STATUS     RESTARTS   AGE
hpe-myweb-78t6p    1/1       Running    0          3m
hpe-myweb-hs0gr    1/1       Running    0          3m
hpe-myweb-jqbjt    1/1       Running    0          3m
hpe-myweb-qcr99    1/1       Running    0          3m
hpe-myweb-t1qh7    1/1       Running    0          3m
```

缩容为 1 个 Pod 副本：

```
# kubectl scale rc hpe-myweb --replicas=1
replicationcontroller "hpe-myweb" scaled

# kubectl get pods
NAME               READY     STATUS     RESTARTS   AGE
hpe-myweb-jqbjt    1/1       Running    0          3m
```

Kubernetes 从 1.2 版本开始引入了一种更加自动化的机制，即根据某些应用程序运行时性能指标的变化，动态调节 Pod 副本的数量。这种全自动的水平扩展（扩容或缩容）机制称

为 Horizontal Pod Autoscaling（HPA）。Kubernetes 当前版本支持基于 CPU 使用率和用户自定义性能指标的自动扩缩容操作，更多相关功能也在快速发展中。

3. 应用版本更新：滚动升级

当集群中的某个微服务版本需要升级时，通常要先停止这个服务所有运行中的实例，再发布新版本的应用程序。如果微服务的实例数量很大，升级工作就变成了一个挑战。Kubernetes 的 RC 提供了滚动升级（rolling update）功能用于解决这个难题。

用户使用 Kubernetes 提供的 kubectl rolling-update 命令即可一键完成应用的版本更新。其实现机制可以简单地描述为：创建一个新的 RC，然后控制旧版本 RC 中的 Pod 副本的数量，使其逐渐减少到 0，同时新版本 RC 中的 Pod 副本的数量从 0 逐步增加到目标值，逐步实现应用版本的升级。在整个更新 Pod 的过程中，总是保证了足够的 Pod 副本数量以提供服务，同时保证服务的不中断，客户端应用几乎无法感知到它访问的服务背后发生的变化。图 5-23 描述了一个具有 3 个 Pod 副本的应用升级的过程。

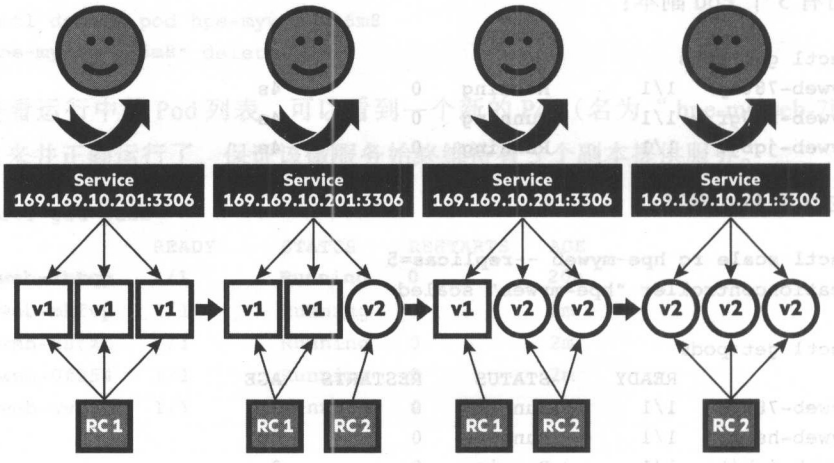


图 5-23　容器应用的滚动升级

下面用 kubectl rolling-update 命令来完成 Pod 的滚动升级，其中 --image 参数指定了新版镜像的名称。滚动升级的执行过程如下：

```
# kubectl rolling-update hpe-myweb --
image=kubeguide/tomcat-app:v2
Created hpe-myweb-d54250d89526fcec590771f79dfa01ae
Scaling                up                hpe-myweb-
d54250d89526fcec590771f79dfa01ae   from   0   to   5,
scaling  down hpe-myweb  from 5  to  0  (keep 5 pods
available, don' t exceed 6 pods)
Scaling   hpe-myweb-d54250d89526fcec590771f79dfa01ae up to 1
```

```
Scaling hpe-myweb down to 4
Scaling   hpe-myweb-d54250d89526fcec590771f79dfa01ae up to 2
Scaling hpe-myweb down to 3
Scaling   hpe-myweb-d54250d89526fcec590771f79dfa01ae up to 3
Scaling hpe-myweb down to 2
Scaling   hpe-myweb-d54250d89526fcec590771f79dfa01ae up to 4
Scaling hpe-myweb down to 1
Scaling   hpe-myweb-d54250d89526fcec590771f79dfa01ae up to 5
Scaling hpe-myweb down to 0
Update succeeded. Deleting old controller: hpe-myweb
Renaming       hpe-myweb       to       hpe-myweb-
d54250d89526fcec590771f79dfa01ae
replicationcontroller "hpe-myweb" rolling updated
```

查看 Pod 列表，可以看到 5 个 Pod 全部更新了：

```
# kubectl get pods
NAME  READY     STATUS    RESTARTS   AGE
hpe-myweb-d54250d89526fcec590771f79dfa01ae-3434k   1/1   Running   0   1m
hpe-myweb-d54250d89526fcec590771f79dfa01ae-j2kq8   1/1   Running   0   1m
hpe-myweb-d54250d89526fcec590771f79dfa01ae-l9k5m   1/1   Running   0   1m
hpe-myweb-d54250d89526fcec590771f79dfa01ae-qfggs   1/1   Running   0   1m
hpe-myweb-d54250d89526fcec590771f79dfa01ae-spf95   1/1   Running   0   1m
```

5.5 Kubernetes 的高级特性

在 Kubernetes 平台上，通过 Pod、RC 和 Service 的组合能够方便地部署微服务，但为了让集群满足更加实际的需求，Kubernetes 还提供了更多的特性用来满足这些需求。本节对 Namespace、ConfigMap、Job 资源对象进行详细描述，说明 Kubernetes 对于一些常见的需求是如何提供支持的。

5.5.1 Namespace 资源隔离

一种常见的工作场景是：在一个组织内部，不同的工作组都要使用服务器资源，按照传统的做法，为不同的工作组购买不同的服务器显然比较浪费。更经济有效的做法是让他们在一个共享的资源池里工作，并且不同组部署的应用应该互不干扰。Kubernetes 使用 Namespace 对不同的工作组进行区分，使得同一个集群的资源可以被共享使用，并且使不同工作组创建的应用相互隔离，如图 5-24 所示。

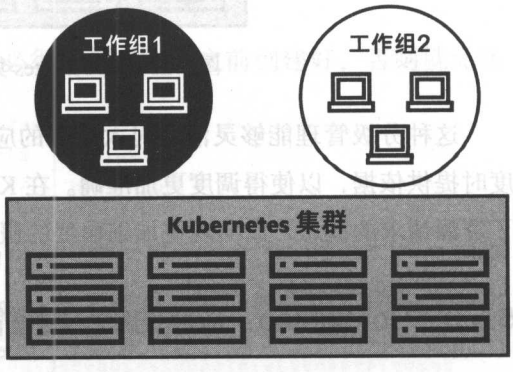


图 5-24 不同工作组共享同一个 Kubernetes 集群

举例来说，一个软件开发团队有两个小组：开发组和测试组。他们在同一个 Kubernetes 集群中同时工作，就可以为他们分别创建名为 dev 和 test 的 Namespace。然后，不同的工作组在其自己的 Namespace 下创建 Pod、RC、Service 等资源对象，这些对象的名字可以相同，因为它们已经被隔离在不同的 Namespace 下了。

5.5.2 容器应用的资源配额管理

在 Kubernetes 集群中运行的实体是大规模的容器应用，对于单个容器以及一组容器进行资源限制就非常必要了。一方面要尽量保证每个容器应用都能正确运行；另一方面要保证整个集群的资源不被少数对资源无限制使用的容器耗尽。在 Kubernetes 系统中，通过容器、Pod 和 Namespace 3 个级别的资源管理来完成集群中各容器应用的资源限制、资源配额管理、资源分配等工作，让整个集群能够健康稳定地运行。

如图 5-25 所示，Kubernetes 在容器、Pod 和 Namespace 3 个级别对集群的资源进行限制和配额管理。在容器级别，主要对 CPU、内存进行限制。在 Pod 级别，可以对一个 Pod 中全部容器的可用资源进行限制。在 Namespace 级别，则可以既对全部 Pod 的计算资源进行限制，也可以对资源对象的数量进行限制，包括可创建的 Pod、RC、Service 等资源对象的个数。

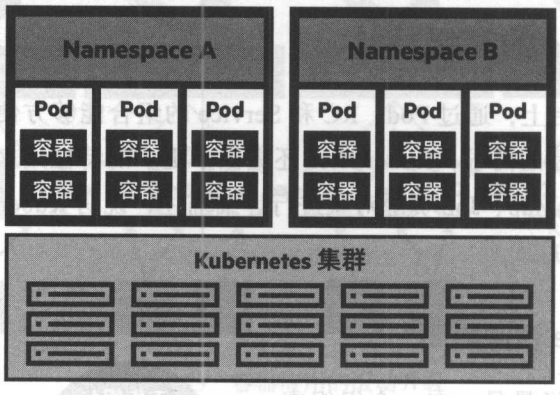


图 5-25 Kubernetes 集群中 3 个级别的资源管理

这种分级管理能够灵活应对大部分的应用场景，同时它还能够为 Master 在进行任务调度时提供依据，以使得调度更加准确。在 Kubernetes 1.2 版本之后，除了资源限制，还提出了资源请求的概念，以实现更加细粒度的服务质量（QoS）等级。

5.5.3 ConfigMap：应用的统一配置管理

为了将应用程序在不同的环境中进行部署和运行，通常要将应用的配置与程序进行分

离，这样应用程序就会更加单纯、复用性更高。在将应用程序容器化之后，配置信息可以通过环境变量或者外挂目录的方式注入容器中。但在大规模容器集群的环境中，多个容器的应用配置管理将变得非常复杂。

Kubernetes 1.2 版本提供了一种统一的应用配置管理方案——ConfigMap，其主要功能是把应用程序所需的配置信息（包括配置文件、配制项、启动参数、环境变量等）放在统一的配置中心进行管理。Kubernetes 在启动容器应用时，再把这些配置信息注入容器中，注入的方式包括：①将 ConfigMag 对象挂载到容器内的某个路径下；②将 ConfigMap 对象映射为容器内的环境变量。ConfigMap 对象以多个 key=value 的形式进行定义，每个 key 值表示一个文件名，value 保存文件的全内容。图 5-26 描述了多个 ConfigMap 被多个容器应用程序使用的场景。

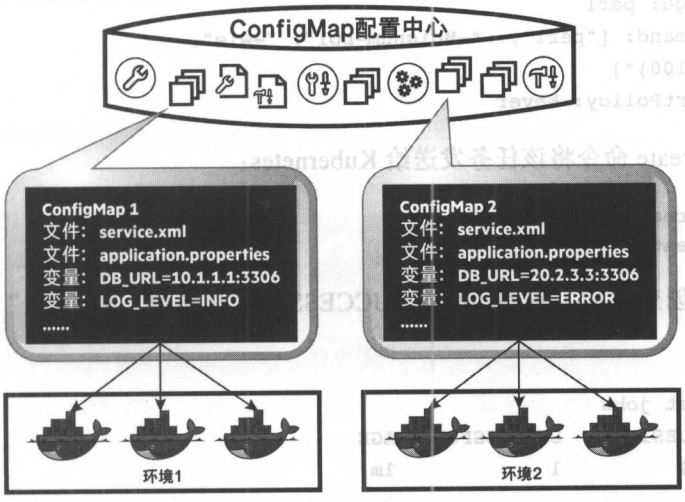


图 5-26 ConfigMap 的配置管理

使用 ConfigMap 的要求是：ConfigMap 对象必须在创建 Pod 之前创建好，否则就无法正确挂载了。

5.5.4 Job：批处理任务

在本章前面的示例中，都是“微服务”类的应用。顾名思义，名为“服务”的应用都是后台长时间运行的应用程序（long-running job）。与之相对的另一种典型应用称为一次性任务或者批处理任务（batch job）。从 Kubernetes 1.2 版本开始，加入了对批处理任务的支持，也就是说，在 Kubernetes 平台上，除了能够运行持续不停的各种微服务应用之外，

也能运行定时任务、并行计算等批处理任务。Kubernetes 引入 Job 资源对象来管理批处理任务。

下例为一个计算 π 值的 Job 的定义（job.yaml）：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle",
"print bpi(100)"]
      restartPolicy: Never
```

用 kubectl create 命令将该任务发送给 Kubernetes：

```
# kubectl create -f job.yaml
job "pi" created
```

在 Job 成功创建之后，查看 Job 的 SUCCESSFUL 状态值，如为“1”则表示 Job 执行成功了。

```
# kubectl get jobs
NAME       DESIRED   SUCCESSFUL   AGE
pi         1         1            1m
```

再看看 Job 执行的结果。首先找到已经结束运行的 Pod，如它的状态为“Completed”则，说明这个任务已经成功执行完毕。如“READY”的状态为 0，则说明这个 Pod 不再是 Running 状态，而是正确结束了。

```
# kubectl get pods --selector=job-name=pi -a
NAME       READY     STATUS      RESTARTS   AGE
pi-37ln5   0/1       Completed   0          2m
```

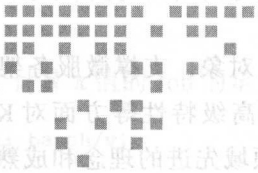
查看容器应用的输出日志，也能看到 Job 的执行结果：

```
# kubectl logs pi-37ln5
3.14159265358979323846264338327950288419716939937510
58209749445923078164062862089986280348253421170 68
```

5.6 总结

本章从Kubernetes的起源、总体架构、核心资源对象，支撑微服务架构的服务发现机制、一个完整微服务系统的部署示例，以及一些高级特性等方面对Kubernetes进行介绍和讲解，希望读者可以一窥谷歌在容器云平台领域先进的理念和成熟的经验。关于Kubernetes的更加完整的实践指导、原理解析、开发运维指南请参考《Kubernetes权威指南》一书。

随着Kubernetes的快速发展，越来越多的新功能已经被开发出来，等待读者去发掘。在Kubernetes社区里，不同模块由不同的特别兴趣组（Special Interest Groups，SIG）主导开发，已经组建了安全、集群联邦、网络、UI、存储、调度算法、大数据支持、Windows容器支持、OpenStack支持、测试、文档等20多个小组，有兴趣和有能力的读者可以申请加入这些小组，贡献自己的力量。



Chapter 6 第 6 章

Mesos

前面几章详细介绍了 Docker 和 Kubernetes，我们可以从中感受到容器技术的强大，尤其是基于微服务的应用架构，更是能够充分发挥出容器技术的优势。

毋庸置疑，容器技术正成为最主流的应用打包标准，但是在工程实践中，我们会发现容器技术并不是万能的银弹（silver bullet），也存在局限性。请考虑下面几个场景是否适合将所有应用都迁移到容器中运行：

1）以 Hadoop/Spark 为代表的大数据类应用。这类应用本身对资源消耗很高，而且自带资源调度管理功能，如果直接迁移至容器环境运行，则无法发挥出容器轻便灵活的特性。

2）对性能要求很高，在物理主机上运行非常稳定的应用。

3）在老旧系统上运行，没有进一步业务发展需求而且改造风险很大的应用。

上面 3 种场景常常出现在实际环境中，那么此时我们的系统可能无法完全容器化，而将要面对容器化应用和非容器化应用并存的环境。

对容器化应用进行资源统一调度管理，有很多强大的工具，如 Kubernetes、Docker Swarm 等；对于非容器应用，以大数据应用为例，有 Hadoop YARN 这样的资源管理框架。那么，对于容器应用和非容器应用并存的环境，该如何进行统一的资源管理呢？

这个问题几种常见解决思路如下。

1）静态分区：将容器化环境和非容器化环境的主机进行静态分割，各自单独管理，相互隔离。

2）虚拟机资源池：通过 OpenStack 等虚拟机技术将底层物理资源虚拟化为资源池，容器环境和非容器环境在其上进行资源的动态调整，以期提高资源利用率。

3）使用 Hadoop YARN 作为统一资源调度平台：将所有非大数据类的应用都改造到 YARN 上面，以 YARN 作为资源调度的核心来对所有资源进行统一管理。

上述方式各有局限性：静态分区本质上还是各自独立运行，无法进行统一管理，也不利于提升资源利用率；虚拟机资源池的方式实施成本高、过程繁琐、管理复杂，对管理运维提出了很大的挑战，性价比较低；在第三种方式中，在 Hadoop 2.0 设计阶段，YARN 就希望成为一个支持所有应用的类型平台，然而由于兼容性包袱等原因，最终 YARN 仍然只对大数据应用支持得比较好，这一领域前几年国内有很多公司在这方面做了探索，成效甚微。

那么是否有一种较为完备的方案呢？这里为大家介绍一种方案——DC/OS（Data Center Operating System），DC/OS 衍生自 Mesosphere 的商业版数据中心操作系统，是一款基于 Apache Mesos 构建的开源数据中心操作系统。DC/OS 的核心目标是让用户可以像使用操作系统一样使用大规模集群数据中心。通过对数据中心的物理资源进行抽象，使用统一的平台对资源进行管理，用户可以使用简单的命令行或者图形界面直接部署、运行和管理各类复杂的分布式应用，包括 Web 应用、NoSQL 数据库、分布式文件系统、Spark 大数据应用等。对于这些应用 DC/OS 提供了一套应用管理工具，使用户可以像使用 Linux 中的 RPM/DPKG 等管理工具一样在数据中心中管理分布式应用。同时该应用管理工具提供可视化 Web 界面和简易命令行两种方式，能满足用户不同的需求。除了应用管理之外，DC/OS 还提供了服务发现、高可用、一致性、健康检查等多种功能，能实现分布式应用的全生命周期管理，构成完善的数据中心管理平台。

上面我们提到的 DC/OS 管理的分布式应用并不局限于容器类应用，还可以直接管理大数据和数据库类应用。那么在 DC/OS 中，大数据应用、容器应用、数据库应用是如何“和谐友好”地运行在一起的呢？不同类型应用之间的资源是如何协调调度呢？

这个问题的答案其实就在于 DC/OS 的内核：Apache Mesos（简称 Mesos）。得益于 Mesos 强大的资源调度管理功能，DC/OS 才能完成上述工作。下面我们将深入学习 Mesos，看一看 Mesos 是如何完成这项工作的。

6.1 Mesos 的背景与概述

6.1.1 Mesos 的产生背景

2005 年 Hadoop 的诞生引领了分布式应用的飞速发展，其后各类分布式应用层出不穷，如图 6-1 所示。Hadoop 作为开源分布式计算框架的鼻祖，是基于谷歌的大数据论文实现的分布式计算基础架构，也是目前应用最广泛的大数据框架。

图 6-1　2005 年 Hadoop 的诞生引领了分布式应用的飞速发展，各类应用层出不穷

Chronos 是一个具备容错特性的作业调度器，是由 Airbnb 公司推出的用来替代 cron（分布式系统中的 cron）的开源产品。我们可以用它来对作业进行编排，支持与 Hadoop 进行交互。

Spark、Storm、Flink、Samza 也都是分布式计算框架，都是 Apache 基金会的开源项目。Flink 与 Spark 类似，都是充分利用内存加速分布式计算的通用分布式计算框架，Samza 和 Storm 则更多用作实时计算的流式处理引擎。

Marathon 是 Mesosphere 公司为 Mesos 开发的，用于在 Mesos 上管理长期服务的应用框架，可以提供对服务的高可用、弹性扩展、故障转移等特性的支持，相当于数据中心的 init.d。

Kubernetes 是由谷歌公司主导的开源的、面向服务的容器编排管理工具。

2005 年，Apache 基金会基于谷歌公开出来的大数据论文实现了 Hadoop 开源项目，从那时起大数据和云计算的概念从硅谷席卷全球。为了满足不同的场景和需求，除 Hadoop 外的其他各类分布式应用也发展壮大起来，例如，Spark、Flink 使用内存加速数据处理，Storm、Samza 擅长于流式数据处理。同时，Kubernetes、Marathon 等侧重于服务管理的 PaaS 基础框架也迅速发展状大。各种不同的分布式应用蓬勃发展是很好的事情，但是对于系统管理人员来说，这也带来了一个很大挑战：不同的分布式应用有不同的特点，但没有一个通用的基础框架能够统一管理和运行这些分布式应用。实现一个集群中同时运行多种分布式应用的基础框架成为强烈的需求，这种基础框架能带来的好处也非常明显：

- 最大化集群资源利用率。
- 方便不同应用之间共享数据。
- 统一管理不同应用，以降低管理成本。

以图 6-2 为例，当 Hadoop、Cassandra、MPI 3 种应用在同一个数据中心运行时，如果

使用静态分区，每个应用各使用 3 台物理机运行，那么资源利用率很低；如果三者可以在 9 台物理机上动态运行，那么通过“削峰填谷”的方式，可以提高资源利用率，降低集群节点数量，节约物理资源和成本。而正是在这种背景和理念下，Mesos 应运而生。

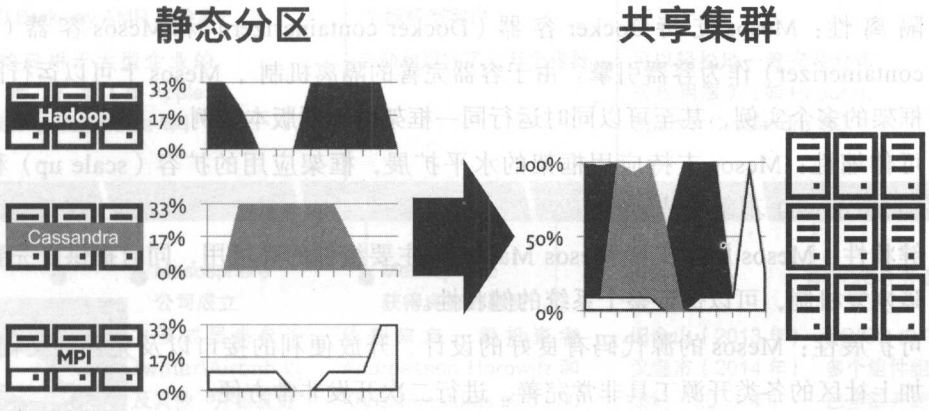


图 6-2 在一个集群中同时运行多种框架可以有效提高集群的资源利用率

6.1.2 Mesos 的特性

Mesos 是为软件定义数据中心而生的操作系统内核。我们可以这样理解：Mesos 的核心功能是实现数据中心资源的统一管理，为运行在 Mesos 上的软件提供资源，这些软件在 Mesos 架构中称为应用框架（framework）。

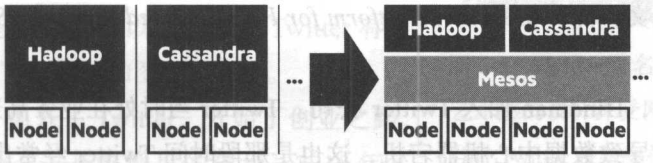


图 6-3 Mesos 工作模型

如图 6-3 所示，在这样的一个典型场景中，使用 Mesos 之前 Hadoop 和 Cassandra 各自运行在两个节点上；使用 Mesos 之后，Mesos 将这 4 个节点的资源全部管理起来，Hadoop 和 Cassandra 作为应用框架运行在 Mesos 之上，使用 Mesos 提供的资源工作。在这个过程中，Mesos 将主机上的资源封装为资源邀约（offer），根据分配算法将资源邀约发送给某一个应用框架，该应用框架收到资源邀约之后根据自己内部的状态和算法决定如何使用该资源邀约。

通过这种两层资源调度模型，Mesos能很好地解决多框架共存的问题。与此同时，Mesos还具有以下特性。

- 高效性：根据Twitter公开的数据，使用Mesos能使数据中心的资源利用率提高约20%。
- 隔离性：Mesos支持Docker容器（Docker containerizer）和Mesos容器（Mesos containerizer）作为容器引擎。由于容器完善的隔离机制，Mesos上可以运行同一个框架的多个实例，甚至可以同时运行同一框架的不同版本实例。
- 可伸缩性：Mesos支持应用框架的水平扩展，框架应用的扩容（scale up）和缩容（scale down）不影响其运行。
- 健壮性：Mesos原生支持Mesos Master等主要组件的高可用，同时提供了完善的故障恢复机制，可以保证整个系统的健壮性。
- 可扩展性：Mesos的源代码有良好的设计、开放便利的接口以及完善的文档支持，加上社区的各类开源工具非常完善，进行二次开发非常方便。

总之，Mesos是一个健全而完善的方案。在下面的内容中，我们会深入地分析Mesos的发展历程和工作机制，来看一下Mesos是如何实现这些强大的特性的。

6.1.3 Mesos的发展历程

2009年，加州大学伯克利分校（UC Berkeley）AMP实验室（AMP Lab）[㊀]的博士生Benjamin Hindman根据AMP实验室的资源分配算法（DRF算法[㊁]）论文实现了Mesos最初的版本。这个初始版本是由C++语言实现的，包含了超过两万行代码，与之对比，最新版本的Mesos已包含超过24万行代码。Hindman在证明Mesos资源调度的先进性之后，将其研究成果发表成论文，即《*Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*》。[㊂]

博士毕业之后，Hindman加入Twitter公司。Twitter当时处在业务高速增长期，系统压力的剧烈波动经常导致数据中心机器宕机，这也是那段时间Twitter经常出现“失败鲸”（fail whale）[㊃]问题的主要原因。

图6-4是Mesos的主要发展历程，供大家参考。

㊀ 详细内容参见https://amplab.cs.berkeley.edu。

㊁ 详细内容参见https://amplab.cs.berkeley.edu/publication/dominant-resource-fairness-fair-allocation-of-multiple-resources-types/。

㊂ 详细内容参见https://amplab.cs.berkeley.edu/publication/dominant-resource-fairness-fair-allocation-of-multiple-resources-types/。

㊃ 详细内容参见http://www.whatisfailwhale.info/。

图 6-4 Mesos“编年史”

为了解决这一问题，Twitter 采取了增加机器、扩展业务节点的措施，但这种单纯增加节点数量的做法导致了数据中心资源利用率低下、资源浪费、成本剧增等问题。为了完全解决这一系列问题，Hindman 将 Mesos 引入 Twitter 的生产环境中，使用 Mesos 管理 Twitter 数据中心上千台物理主机，将 Twitter 的数据中心资源利用率提高了 20% 以上，帮助 Twitter 有效地解决了“失败鲸”问题，如图 6-5 所示。

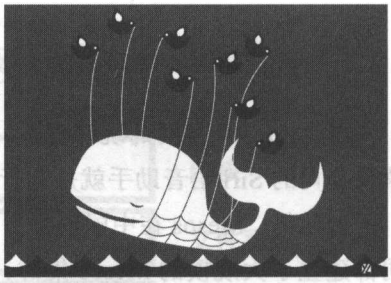


图 6-5 Twitter 著名的“失败鲸”，富有艺术感的设计深受网友好评，背后却是当时 Twitter 系统架构所面临的严峻问题

之后 Hindman 离开了 Twitter，开始了创业之路，同时积极投身于 Mesos 开源项目的建设。2010 年年底 Mesos 开源项目正式成为 Apache 的孵化项目（Apache Incubator）。2013 年 Mesos 孵化成功，升级为 Apache 顶级项目；同年，Hindman 创立商业公司 Mesosphere，团队的创始人均为来自 Twitter、Airbnb 等硅谷技术型公司，Mesosphere 致力于推动 Mesos 的商业化。2014 年 Mesosphere 商业化产品——基于 Mesos 的数据中心操作系统 DC/OS（商业版）——迎来了第一个客户，成功支撑了上万数量级容器的运行和管理，之后 Mesosphere 拿到 5 000 万美元的融资。2015 年 DC/OS 提出 Universal App 的概念，支持针对 Mesos 环境的一键部署 Hadoop、Spark 、Cassandra 等复杂分布式应用的功能。2016 年，Mesosphere 推出开源版的

DC/OS，开源版 DC/OS 沿袭原商业版 DC/OS 架构，以 Mesos 为内核，包含 30 多种组件，支持一键部署、可视化管理、实时监控等功能，可以称得上是目前市面上最完善、最成熟的开源数据中心管理平台。

如图 6-6 所示，飞速发展的 Mesos 和 DC/OS 不仅吸引了大量的开源爱好者参与其中，更有大量国内外公司将其引入生产实战，商用范围快速扩张。

图 6-6 从 Twitter 到微博，国内外公司在生产环境中大规模使用 Mesos

根据 Mesos 官网的统计显示，目前超过 110 家大型公司在使用 Mesos，最广为人知的苹果公司的 Siri 语音助手就是运行在 Mesos 集群上。

近几年，Mesos 在中国也得到了广泛的应用，豆瓣、爱奇艺、去哪儿、微博、小米等公司都建立了大规模的 Mesos 集群，并分享了 Mesos 的使用经验。目前来看，虽然 Mesos 并没有如同 Kubernetes 一样异常火爆，但是简洁易用的功能、超强的适应性和可扩展性、充分验证过的稳定性和可靠性等优势使得 Mesos 得到了广泛认可，非常适合于生产环境的大规模使用。

蓬勃发展的背后是 Mesos 强大的技术根基，下面的章节我们将从设计思路、软件架构、核心算法等方面对 Mesos 进行技术分析。

6.2 Mesos 的架构与核心

6.2.1 Mesos 的设计与架构

Mesos 是为了解决分布式环境下不同应用资源竞争与分配这个问题而诞生的，其设计

目标如下：

- 支持各类应用（包括已知的和未来未知的各类应用）。
- 强大的扩展能力（目标支持 10 万节点级别的集群）。
- 高可靠性（支持容灾和快速故障恢复）。
- 提升系统资源的利用率。

为实现上述设计目标，Mesos 的架构采用了基于两级调度的微内核设计：Mesos 的核心功能非常简单，只负责资源的信息收集和粗粒度调度分发；而使用 Mesos 提供资源的应用框架，则自行负责资源的细粒度管理和使用。

在这个架构中，分布式环境如同一个传统的操作系统，而 Mesos 承担的角色正是这个分布式操作系统中的内核，对应关系如图 6-7 所示。

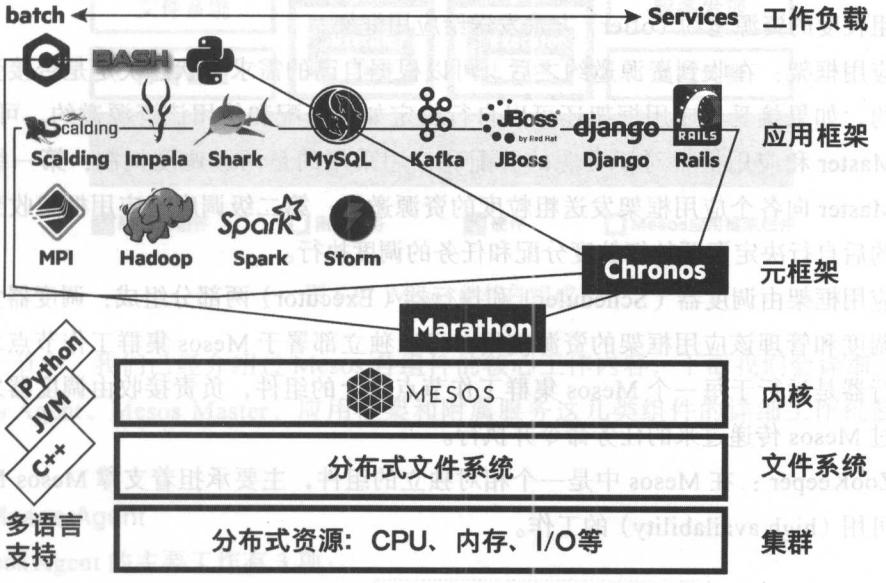


图 6-7 使用 Mesos 构建的分布式数据中心操作系统与传统的 Linux 系统对应

作为分布式数据中心操作系统的内核，Mesos 统一管理所有资源，并对这些资源进行分配和调度。下层是提供资源的基础设施层（分布式文件系统、共享存储、裸机、虚拟机、云等）；上层是使用 Mesos 提供资源的应用框架。应用框架根据功能定位的不同可以分为管理应用框架的元框架和实现具体应用功能的应用框架。例如，Marathon、Chronos 等框架就属于元框架，Marathon 在分布式操作系统中的角色类似于 Linux 操作系统中的 Init.d，提供服务管理功能，Chronos 的角色类似于 cron，提供定时任务和批处理管理功能。其他的应用，如 Spark、Kafka 等，则实现自己的应用功能，各司其职。

在这个架构中，最核心的就是 Mesos 和各类应用框架，应用框架的介绍会在后面的章节中详细展开，下面我们先关注 Mesos 的组成及其工作原理。

如图 6-8 所示，Mesos 的软件主体架构由 Mesos Master、Mesos Agent、应用框架（Framework）和 ZooKeeper 等其他附属组件组成，各组件的功能如下。

- Mesos Agent：是部署到各个工作节点上的 Mesos 组件，其主要功能是负责收集工作节点信息并上报给 Mesos Master，这些信息包括资源信息（如 CPU、网络端口、存储等）、应用框架信息（如应用框架的运行状态、资源使用情况等）、任务信息（如任务调度和执行情况等）等。Agent 还能根据接收到的 Master 下发的任务来执行相应操作。
- Mesos Master：Mesos 的核心，在收集并汇总 Agent 上报的信息之后，Master 可以知道整个集群的所有运行状态。Master 会根据内部算法来选择将集群中的空闲资源以粗粒度的资源邀约（offer）方式发送给应用框架。
- 应用框架：在收到资源邀约之后，可以根据自己的需求和状态决定是否接受资源邀约，如果接受，应用框架还可以自行决定如何分配和使用该资源邀约。可以看出，Master 和应用框架对于资源的分配和使用采用的是两级调度机制：第一级调度是 Master 向各个应用框架发送粗粒度的资源邀约；第二级调度是应用框架收到资源邀约后自行决定资源的细粒度分配和任务的调度执行。
- 应用框架由调度器（Scheduler）和执行器（Executor）两部分组成：调度器负责统一调度和管理该应用框架的资源使用，可以独立部署于 Mesos 集群工作节点之外；执行器是运行于每一个 Mesos 集群工作节点之上的组件，负责接收由调度器发出并通过 Mesos 传递过来的任务命令并执行。
- ZooKeeper：在 Mesos 中是一个相对独立的组件，主要承担着支撑 Mesos Master 高可用（high availability）的工作。

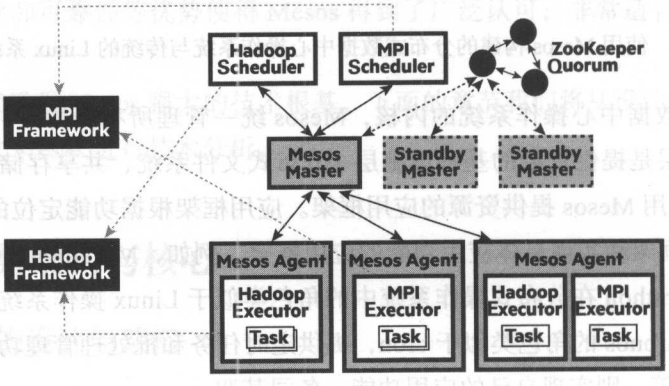


图 6-8 Mesos 的软件主体架构

6.2.2 Mesos 系统组件

如图 6-9 所示，中间部分对应 Mesos Master 和 Mesos Agent；两侧是 Mesos 的附属服务，提供监控、部署管理、服务发现、负载均衡、文件系统、一致性等功能；下面的灰色部分是硬件基础设施层，可通过裸机私有云或者公有云来支撑；最上面则是各个应用框架。

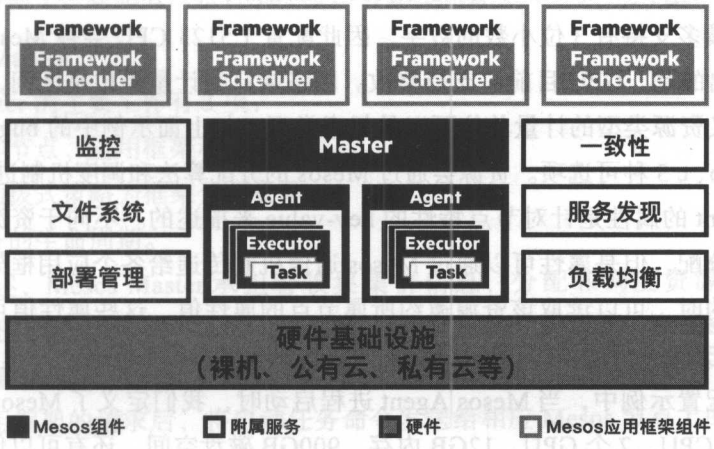


图 6-9 Mesos 的软件组成

在上节中，我们已经介绍过 Mesos 各组件的核心工作内容，下面我们会详细分解和介绍 Mesos Agent、Mesos Master、应用框架和附属服务这几类组件的详细工作机制和配置方法。

1. Mesos Agent

Mesos Agent 的主要工作有 3 项：

- 收集节点的信息，包括节点的资源信息、任务状态和资源使用信息、应用框架的资源信息和状态信息，并将这些信息上报给 Mesos Master。
- 负责利用现有资源执行 Mesos Master 下发的任务。
- 提供隔离机制以保证资源隔离，并确保任务获得准确的资源份额。

Mesos Agent 的曾用名为 Mesos Slave，后来改名为 Mesos Agent（截至 Mesos 0.28.2 版本，部分 Mesos 命令中仍然会出现“Slave”字样，后续版本会陆续改进）。

Mesos Agent 有两个主要配置项：资源（resource）和属性（atrributes）。一个典型的 Mesos Agent 启动参数配置如下：

```
--resources='cpus:30;gpus:2;mem:122880;disk:921600;ports:[80,21000-
```

```
29000];bugs:{a,b,c}'
--attributes='rack:rack-2;datacenter:china;os:centosv7.2'
```

Mesos Agent 的资源配置支持 5 种内置资源类型和其他的扩展资源类型。5 种内置资源类型分别是 CPU、GPU、内存、磁盘和网络端口；扩展资源类型则是由用户自定义的资源类型。Mesos 的资源类型计量单位分为 3 种：标量、区间和集合。CPU、GPU、内存和磁盘的计量单位是标量（数字标量），CPU 和 GPU 的默认单位是核心数（cores），内存的默认单位是兆字节（megabytes，MB），磁盘的默认单位是吉字节（gigabytes，GB）。Mesos 中的标量计量单位最多支持有 3 位小数的数字，因此配置 1.5123 CPU 会被 Mesos 识别为 1.512 CPU，特别指出的是，GPU 目前只支持整数。网络端口的计量单位是区间，区间可以指定为多段。自定义资源类型的计量单位可以是任意类型，如上面示例中的 bugs 的计量类型是集合，包含 a、b、c 3 种可选项。资源会通过 Mesos 的分配算法和调度机制进行分配和调度。

Mesos Agent 的属性是针对节点特性的 key-value 来描述的。不同于资源，属性不会直接参与调度与分配，但是属性可以通过 Mesos 通信机制传递给各个应用框架，应用框架在接收到资源邀约时，可以读取该资源邀约所属节点的属性值，这些属性值可以指导应用框架对节点特性进行区分和识别，为应用框架自身的调度提供参考依据。

在上面的配置示例中，当 Mesos Agent 进程启动时，我们定义了 Mesos 可以使用这台机器上的 30 个 CPU、2 个 GPU、12GB 内存、900GB 磁盘空间，还有可以使用的端口范围（80 端口和 21000 ～ 29000 端口段），最后是这台机器有哪些 Bugs。而在属性配置中，我们为这个 Mesos 节点配置了以下属性：机架（rack）是 2 号机架（rack-2），数据中心（data center）是中国，操作系统是 CentOS 7.2。

Mesos Agent 在收集信息的时候，将上述信息中的资源信息作为该节点可以提供给 Mesos 集群使用的资源的基本信息，对于一些未手动设置的资源信息，Mesos Agent 会尝试自动获取该节点的全部可用资源数量作为资源信息。在大部分环境中，其他类型的资源都能得到很好的支持，但是由于 GPU 资源隔离是由主机显卡驱动来支持的，因此 GPU 资源不能完全保证在所有环境下都能正常工作。此外，Mesos Agent 也会将节点属性透明传递给 Mesos Master 和应用框架。

资源和属性的区别可以结合下面这个案例来说明：一个大型的跨网络 Mesos 集群，其节点分布在北京机房和上海机房，假设现在的节点为北京节点 1（资源配置为 5 CPUs 属性配置为 rack：beijing，状态为空闲）、上海节点 1（资源配置为 25 CPUs，属性配置为 rack：shanghai，状态为空闲）、上海节点 2（资源配置为 4 CPUs，属性配置为 rack：shanghai，状态为使用中），而运行于 Mesos 之上的应用框架 A 需要 3 CPUs，同时由于该应用为上海地区提供服务，因此应用框架希望能使用上海机房的节点。

那么一个可能的 Mesos 调度过程是这样的：上海节点 1 上的资源余量很大，但是此时

Mesos Master 调度算法基于公平原则（下文会详细介绍），并没有将上海节点 1 的资源邀约直接发给应用框架 A，而是发给了别的应用框架，此后 Mesos 将北京节点 1 的资源邀约发送给应用框架 A；应用框架 A 在收到该资源邀约后，发现该资源的属性为 rack：beijing，判断该资源所属节点在北京，因此拒绝这个资源邀约；此后上海节点 2 上的任务执行完成，资源被释放出来，Mesos 将上海节点 2 的资源邀约发送给应用框架 A，A 判断上海节点 2 的资源符合所有条件，接收该资源邀约并开始使用。

通过上面的例子，我们可以很直观地看出资源与配置在 Mesos 中的差异。

2. Mesos Master

Mesos Master 的主要工作有 3 项：

- 收集集群节点、应用框架和任务的信息。
- 根据可插拔式策略为框架分配资源。
- 管理任务的生命周期。

如前文所述，Mesos Master 承担着收集集群信息、分配和调度资源的职能，同时 Mesos Master 还会管理运行于 Mesos 节点上的各个任务的生命周期：在 Mesos 体系中，应用框架并不能直接对任务进行控制操作，而必须通过 Mesos Master 来管理任务，Mesos Master 收到应用框架的请求后，将启动任务命令发送给相应 Mesos 节点上的 Mesos Agent，Mesos Agent 在接收到命令之后，创建隔离资源的运行环境（沙箱），并使用命令参数调用应用框架的执行器，开始执行任务；同时，Mesos Agent 会定期收集任务信息上报给 Mesos Master，Mesos Master 收集到任务信息后会根据运行情况决定对任务的处理，如通知清理任务、重启任务或者强制终止任务等。Mesos 任务的完整生命周期如图 6-10 所示。

图 6-10 Mesos 任务的完整生命周期

3. 应用框架

应用框架是运行在 Mesos 上的分布式应用，使用的是 Mesos Master 给它提供的资源。一个完整的应用框架是由调度器（Scheduler）和执行器（Executor）组成的。调度器负责处理 Mesos Master 的资源邀约、调度应用框架自身的资源分配和任务分配；执行器则部署在各个 Mesos 节点上，通过多种方式完成任务的执行，可以在一个执行器中运行一个任务，也可以在一个执行器中使用多进程来完成多个任务。

应用框架的功能包括资源分配、应用调度、任务管理、框架 API 管理，以及提供调度器和执行器的通信功能。应用框架需要管理该应用的资源分配和调度执行，同时需要实现与 Mesos Master 的通信，以及 Mesos 要求的框架 API。

图 6-11 是 Mesos 应用框架的功能结构图，Mesos Master 发送资源邀约给应用框架的调度器，而调度器通过发送请求到 Mesos Master 再到 Mesos Agent 最后到 Executor 的方式来控制自身任务的调度和执行。

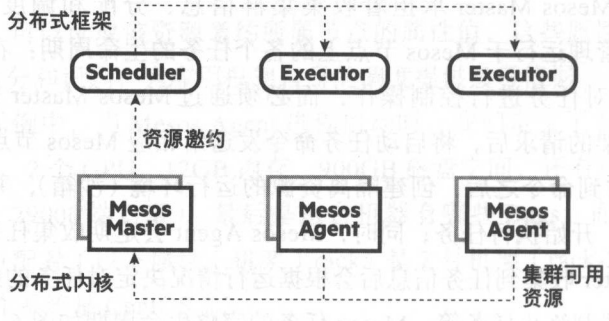


图 6-11　Mesos 应用框架的功能结构

4. 附属服务

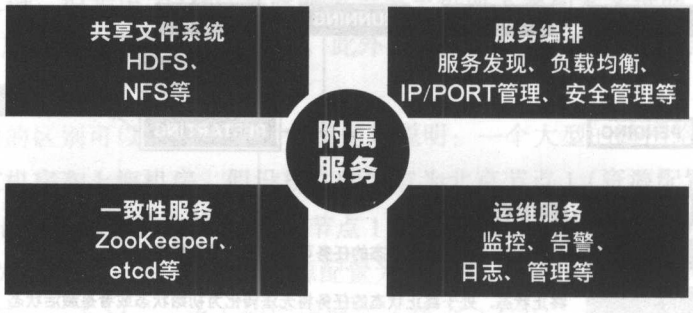


图 6-12　Mesos 的主要附属服务

Mesos 集群除了 Mesos Master、Mesos Agent 和应用框架之外，还需要一系列的附属服

务对整个集群的运行提供支撑，如图 6-12 所示。Mesos 集群中常见的附属服务包括：

- 共享文件系统：提供 Mesos 可用的分布式文件系统，以统一视角对待资源，而不用考虑物理环境的差异。
- 一致性服务：ZooKeeper 和 etcd 为 Mesos 提供分布式环境的高可用支撑。
- 服务编排：在 Mesos 的基础上，为应用框架提供服务发现、负载均衡、网络资源管理、安全管理等应用层的管理功能。这些功能可以结合 Mesos 提供的基础资源管理，用于构建一个完整的 PaaS 平台。
- 运维服务：提供监控、告警、日志、管理等功能，是一个完善的数据中心操作系统的基础保障。

这些附属服务不属于 Mesos，但是是集群稳定运行的基础，能帮助我们更好地管理 Mesos 集群。

我们在本节系统地介绍了 Mesos 的功能组件，本章最后我们来深入学习 Mesos 的核心调度机制。

6.2.3 Mesos 的调度算法

1. Mesos 两级调度

前文我们提到了 Mesos 的资源调度分为 Master 粗粒度的资源分配和应用框架自身的细粒度调度，这正是 Mesos 最有名的两级调度机制。

图 6-13 展示了一个典型的两级调度流程，分解说明如下。

1）运行于 Mesos 节点 s1 之上的 Mesos Agent 1 向 Mesos Master 上报 s1 的空闲可用资源为：4 CPU、4 GB 内存。

2）Mesos Master 收到该信息后，经过资源分配策略模块的分配算法计算后，Mesos Master 选择将该资源以资源邀约的形式发送给 Framework 1。该资源邀约中包含了 s1 的资源信息、属性信息和其他信息。

3）Framework 1 在接收到 Mesos Master 发送过来的资源邀约之后，根据应用框架自身的运行状态和策略选择是否接收该资源邀约。在本例中，Framework 1 决定接收该资源邀约，那么此时 Framework 1 可以使用 s1 上的资源。之后 Framework 1 的调度器根据自身的调度策略决定如何划分和使用 s1 上的资源。在本例中，Framework 1 的调度器决定将 s1 上的资源分成两份，每份运行一个任务，具体的用法如图 6-13 所示：task1 使用 2 CPU 和 1 GB 内存，task2 使用 1 CPU 和 2 GB 内存。Framework 1 将这个决策发送给 Mesos Master。

4）Mesos Master 收到 Framework 1 发送过来的信息，确认调度任务的目标机器信息后，给目标机器 s1 上的 Mesos Agent 即 Agent 1 发送信息，通知 Agent 1 执行相应的任务。而

Agent 1 在接收到 Mesos Master 的信息后，根据 Mesos Master 信息里的要求创建两个隔离的资源环境，使用 Framework 1 的调度器提供的参数来调用 Framework 1 的执行器，执行器利用该资源环境分别执行任务 task1 和 task2。

5）此后，s1 上仍然剩余 1 CPU 和 1 GB 内存，那么 Agent 1 会继续将该资源上报给 Mesos Master，不断重复步骤 1 ～ 4 的流程。

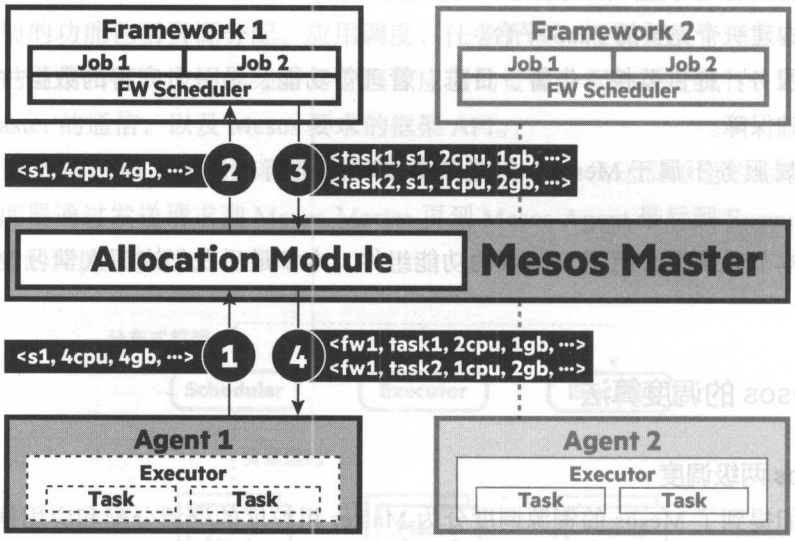


图 6-13　Mesos 的两级调度模型

在上述过程中，Mesos Master 决定将 s1 的资源发送给谁（本例中 Mesos Master 决定选择 Framework 1，而不是 Framework 2），这是第一级调度；而在 Framework 1 接收资源后，由其调度器根据 Framework 1 的运行状态和策略决定启动两个任务，并给出每个任务使用资源的详细数值（task 1 <2 CPUs, 1 GB RAM>，task 2 <1 CPUs, 2 GB RAM>），这就是第二级调度。

总结一下：在 Mesos 的两级调度中，第一级调度是粗粒度的调度，由 Master 将某个节点的全部可用资源以资源邀约的形式发送给某一个应用框架，第二级调度是细粒度的调度，由接收到资源邀约的应用框架来自行决定如何分配和使用该资源。

2. DRF 算法

Mesos 的两级调度是一个简洁、优雅而又能满足各类复杂需求的调度模型。其中第二级调度如何完成它的工作，完全取决于应用框架自身的需求和实现方式，那么在关键的第一级调度中，Mesos 又是如何来选择向哪个应用框架发送资源邀约的呢？

答案就是 Mesos 调度模块的分配算法。这个分配算法会根据 Master 收集到的资源信息

和应用框架信息，来选择一个应用框架作为本次资源邀约发送的目标。在 Mesos 中，这个调度模块的分配算法是开放的，我们可以选择加载自定义的分配算法，与此同时，Mesos 为我们提供了一个非常强大的默认算法——DRF（Dominant Resource Fairness）算法。

DRF 算法是 AMP 实验室为了解决在多维度资源类型场景下进行资源竞争时，如何进行公平、有效的分配这个问题而提出来的一种通用的公平算法。

为了更好地理解 DRF 算法，我们先看一种更简单的算法——最大最小公平（Max-Min Fairness）算法。最大最小公平算法是非常经典的公平算法，常用于单一类型资源的分配，如 Hadoop 的公平调度器和容量调度器、网络路由中的 wfq 调度和 cfsq 调度、操作系统的 rr 调度和 cfs 调度等。最大最小公平算法的形式化定义如下：

- 资源按照需求递增的顺序进行分配。
- 不存在用户得到的资源超过自己的需求的情况。
- 未得到满足的用户等价的分享资源。

我们结合下面这个例子（如图 6-14 所示）来看一下最大最小公平算法是如何工作的。

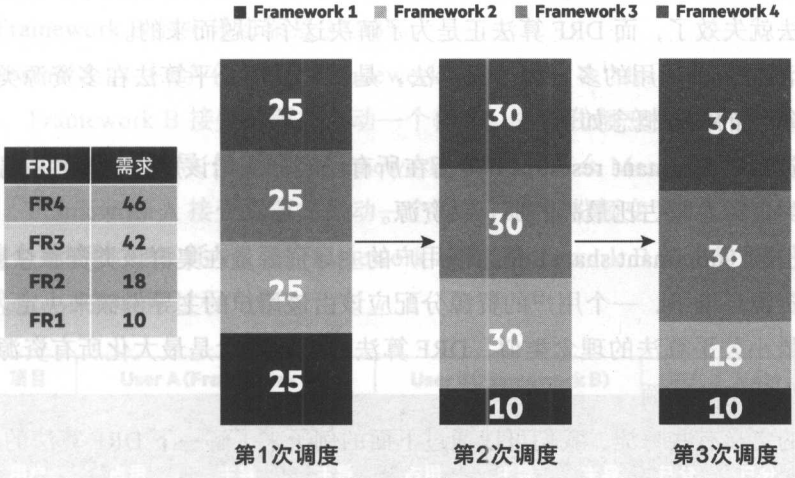


图 6-14 最大最小公平算法的调度过程

系统可用资源为 100 GB 内存，4 个应用的需求分别为 10 GB、18 GB、42 GB 和 46 GB 内存（已按照资源需求进行递增排序），总需求为 116 GB 内存，显然系统无法同时满足所有需求。使用最大最小公平算法进行资源调度：

1）第一次调度逻辑处理之后，4 个应用平均分配 100 GB 的内存，每个应用占用 25 GB，此时应用 1 的需求是 10 GB，因此退回其多余的 15 GB 内存，交由系统分配给未满足的其他 3 个应用。

2）第二次调度后，应用 2、3、4 平均分配应用 1 退回的 15 GB 内存，每个应用加 5

GB 的内存，最后的分配形式为每个应用 30 GB 的内存，此时应用 2 超过了其需求 18 GB，退回多余的 12 GB 内存，交由系统分配给未满足的其他两个应用。

3）第三次调度时，应用 3、4 平均分配应用 2 退回的 12 GB 内存，每个应用加 6 GB 的内存，最终应用 3、4 均占用 36 GB 的内存，而应用 1、2 则分别维持 10 GB 和 18 GB 不变。此时，系统中应用 3 和应用 4 未满足其需求，两者平分资源，而应用 1 和应用 2 已经满足其需求，且不超过应用 3 和应用 4 的占用资源。

可以看出经过上述算法分配之后，最终结果满足最大最小公平算法的形式化定义。

最大最小公平算法最大化了所有资源得不到满足的用户中最小的分配资源数量。因此，最大最小公平算法被认为是一种很好地权衡了效率和公平的资源分配算法，在数学上也可以证明用最大最小公平算法得到的资源分配是帕累托最优（Pareto optimality）的（意味着在公平的基础上，我们不可能在不损害其他用户利益的前提下去改善某个用户的资源分配）。

不过仔细观察上述场景不难发现，最大最小算法处理的只是单一资源类型的分配问题，如上例中，资源类型只有内存这一种。而 Mesos 则需要面对多种资源类型（如 CPU、内存、磁盘、网络端口、GPU 等）。这时候，由于无法直接比较两种资源的大小，所以简单的最大最小公平算法就失效了，而 DRF 算法正是为了解决这个问题而来的。

DRF 算法是一种通用的多资源分配算法，是最大最小公平算法在多资源类型情况下的演化。DRF 算法的核心概念如下：

- 主导资源（dominant resource）是指在所有已经分配给该用户的多种资源中，在集群中该类资源总量占比最高的那一项资源。
- 主导份额（dominant share）是指该用户的主导资源量在集群该类资源总量中的占比。
- 在多资源环境下，一个用户的资源分配应该由该用户的主导份额来决定。

与最大最小公平算法的理念类似，DRF 算法的核心理念是最大化所有资源得不到满足的用户中最小的主导份额。

形式化的定义有些晦涩，我们可以通过下面的例子来了解一下 DRF 算法的工作流程。

场景：

- 系统总共可分配资源：<9 CPU，18 GB 内存 >。
- Framework A 希望执行任务，每个任务需要的资源为 <1 CPU，4 GB 内存 >。
- Framework B 希望执行任务，每个任务需要的资源为 <3 CPU，1 GB 内存 >。

通过 DRF 算法，最终 Framework A 和 Framework B 各自会拿到多少资源，启动多少个任务呢？下面我们通过 DRF 算法的分解步骤来了解一下 DRF 算法是如何进行资源分配的。

根据上面的定义，主导资源是指用户使用资源在所有资源中占比最高的那一类 。对于 Framework A 而言，无论启动多少个任务，其内存占用率（$n \times 4/18$）都会比其 CPU 占用率（$n \times 1/9$）高，因此 Framework A 的主导资源为内存。反之，Framework B 的主导资源为

CPU。相应地，当 Framework A 和 Framework B 只启动一个任务时，各自的主导份额分别是：Framework A 的内存占比为 4/18，Framework B 的 CPU 占比为 3/9。

DRF 算法的核心目标是让不同用户的主导份额尽量保持平衡，结合图 6-15 中的流程，我们来看一下如何保持主导份额的平衡。

1）初始状态是零，Framework A 和 Framework B 都没有启动任务，此时 Framework A 和 Framework B 的主导份额都是 0%，DRF 算法随机选一个作为资源分配的目标，假设 DRF 算法选择了 Framework B 并发出资源邀约，那么 Framework B 接收该资源邀约后启动一个任务。此后 Framework B 占用的资源为 <3 CPU,1 GB>，主导资源为 CPU，主导份额是 3/9（33.33%）。这个时候整个系统还有 6 个 CPU 和 17 GB 的内存。

2）DRF 算法发现 Framework B 的主导份额比 Framework A 高，基于公平的原则，此时 DRF 算法会把资源邀约发送给 Framework A。Framework A 接收资源邀约后启动一个任务，它的主导资源是内存，主导份额变为 4/18（22.22%）。

3）DRF 算法发现 Framework B 的主导份额仍然比 Framework A 高，这时 DRF 算法还是会把资源继续发送给 Framework A，此后 Framework A 再启动一个任务，主导份额变为 44.44%，而 Framework B 的主导份额是 33.33%。

4）Framework A 的主导份额比 Framework B 高，那么 Mesos 的资源会发送给 Framework B，Framework B 接受资源并启动一个任务，主导份额变为 6/9（66.66%）。

5）Framework B 的主导份额比 Framework A 高，那么 Mesos 的资源会发送给 Framework A，Framework A 接受资源并启动一个任务，主导份额变为 12/18（66.66%）。

此后，整个系统的 CPU 耗尽，而 Framework A 和 Framework B 的主导份额达到了一个平衡的稳定状态，本次调度过程完成。

项目	User A (Framework A)			User B (Framework B)			CPU	RAM
信息说明	每个任务占用的资源为 < 1 CPU，4 GB >			每个任务占用的资源为 < 3 CPU，1 GB >			9cores	18 GB
用户选择	占用资源	主导资源	主导份额	占用资源	主导资源	主导份额	已分配额	已分配额
0.初始	0/9,0/18	0	0%	0/9,0/18	0	0%	0/9	0/18
1.User B	0/9,0/18	0	0%	3/9,1/18	3/9 CPU	33.33%	3/9	1/18
2.User A	1/9,4/18	4/18 RAM	22.22%	3/9,1/18	3/9 CPU	33.33%	4/9	5/18
3.User A	2/9,8/18	8/18 RAM	44.44%	3/9,1/18	3/9 CPU	33.33%	5/9	9/18
4.User B	2/9,8/18	8/18 RAM	44.44%	6/9,2/18	6/9 CPU	66.66%	8/9	10/18
5.User A	3/9,12/18	12/18 RAM	66.66%	6/9,2/18	6/9 CPU	66.66%	9/9	14/18

图 6-15 DRF 算法进行任务调度的流程

需要特别指出的是，在上述调度过程中，如果有两个用户的主导份额相同，DRF 算法

会如何挑选呢？答案是随机。可以简单证明：除了某些极端的边界条件之外，这种随机挑选几乎完全不会影响 DRF 算法的分配结果。

通过上述过程可以看出，DRF 算法的目标是平衡用户间的主导份额，也可以说 DRF 算法就是以主导份额为单一资源类型的最大最小公平算法的演进。在数学上可以证明 DRF 具有以下几个特点。

- 策略可验证 (strategy proofness)：用户无法通过夸大自己的需求来获得更多的优势。
- 鼓励资源共享 (sharing incentive)：DRF 算法保障了每个用户所能获得的最少资源，同时提高了用户可能拿到的最大资源，所以用户更愿意使用 DRF 算法，而不是独占 1/*N* 的资源。
- 单一资源公平 (single resource fairness)：针对一种资源 DRF 算法退化为最大最小公平算法，单一资源两者等价。
- 无嫉妒 (envy free)：每个用户获得的资源都不比其他用户差，这也意味着不同用户的相同请求会得到相同的分配。
- 瓶颈公平性 (bottleneck fairness)：当两个用户的同一类资源成为瓶颈时，DRF 算法等价于最大最小公平算法。
- 单调性 (monotonicity)：在系统中添加资源或减少用户只会增加剩余用户所获得的资源。
- 帕累托最优：意味着在公平的基础上，我们不可能在不损害其他用户利益的前提下去改善某个用户的资源分配。

DRF 算法具备上述诸多优点，非常契合 Mesos 面临的多维度资源分配场景，因此当仁不让地成为 Mesos 的首选算法。

不过在实际应用中，Mesos 经常会面临应用框架优先级有区别的场面，因此需要对 DRF 算法加以强化，以处理应用不同优先级的问题，这种强化就是让 DRF 算法加入对优先级权重的支持，相应地，DRF 算法就升级为加权 DRF 算法。

3. 加权 DRF 算法

加权 DRF（weighted Dominant Resource Fairness，wDRF）算法是指 DRF 在计算用户主导资源的份额时会先除以权重（weight）再计算为最终加权主导份额的算法。

图 6-16 这个例子很直观地说明了加权 DRF 算法的计算方式：

- 全部资源为 <9 CPU,18 GB>。
- User A 的权重为 2，每个任务使用的资源为 <1 CPU,4 GB>。
- User B 的权重为 1，每个任务使用的资源为 <3 CPU,1 GB>。

项目	User A (Framework A)			User B (Framework B)			CPU	RAM
信息说明	任务占用的资源为 < 1 CPU，4 GB >，权重为 2			任务占用的资源为 < 3 CPU，1 GB >，权重为 1			9cores	18 GB
用户选择	占用资源	主导资源	加权份额	占用资源	主导资源	加权份额	已分配额	已分配额
0.初始	0/9,0/18	0	0%	0/9,0/18	0	0%	0/9	0/18
1.User B	0/9,0/18	0	0%	3/9,1/18	3/9 CPU	33.33%	3/9	1/18
2.User A	1/9,4/18	4/18 RAM	11.11%	3/9,1/18	3/9 CPU	33.33%	4/9	5/18
3.User A	2/9,8/18	8/18 RAM	22.22%	3/9,1/18	3/9 CPU	33.33%	5/9	9/18
4.User A	3/9,12/18	12/18 RAM	33.33%	3/9,1/18	3/9 CPU	33.33%	6/9	13/18
5.User A	4/9,16/18	16/18 RAM	44.44%	3/9,1/18	3/9 CPU	33.33%	7/9	17/18
5.User B	3/9,12/18	12/18 RAM	33.33%	6/9,2/18	6/9 CPU	66.66%	9/9	14/18

图 6-16 wDRF 算法进行任务调度的流程

整个调度流程跟上面的 DRF 算法的调度流程基本一致，这里不再赘述，唯一需要注意的是调度的依据由主导份额改为加权份额，而加权主导份额的值 = 主导份额 / 权重。另外需要特别说明的是，图 6-16 中最下面两行都是第五步，由于第四步执行完之后，Framework A 和 Framework B 的加权份额都是 33.33%，因此第五步可能选择 Framework A，也可能选择 Framework B，这是一种特别的边际情况。

通过 wDRF 算法，我们可以实现支持带权重的资源分配。在 Mesos 的配置中，我们可以利用 weights（权重）和 roles（角色，更详细的说明参见 6.2.4 节）参数来设置应用框架的权重（如图 6-17 所示）。

- role 名称后面接权重，形如 role1=weigth1。
- 权重值可以是整数，也可以是小数。

为不同的应用框架设置不同的权重之后，就可以将资源按照相应的比例倾斜分配。

```
--roles="spark,marathon"
--weights="spark=2.5,marathon=1.5"
```

图 6-17 Mesos Master 启动参数中指定了角色和相应的权重

6.2.4 Mesos 的核心机制

1. Mesos 资源分配

Mesos 两级调度中第一级调度采用 wDRF 算法实现，前文详细介绍了 wDRF 算法的诸多优点，但是由于现实应用中场景的复杂性，我们还是会发现 wDRF 算法有一些局限性

- “饿死”：一些大型应用需求一大块资源，而整个系统又是由很多小的、满负载的应用组成的，每一时刻整个系统会释放一小块资源出来，小块资源发送给大型应用之后由于无法满足应用需求而被拒绝，然后迅速被其他应用框架接收并使用，持续的

过程会导致大型应用始终无法拿到足够的资源来运行。例如，系统有 100 个 CPU 全部被使用，每 30 秒释放 1 个 CPU 出来，这个 CPU 所在主机的资源会作为资源邀约发给大型应用框架，但由于该应用框架启动任务需要 50 个 CPU，所以该应用框架只能拒绝这个资源邀约，而随后这个资源邀约被发送给其他小型应用框架，并被小型应用框架使用。假设这种场景一直持续，那么虽然整个系统不断有 CPU 资源被释放出来，但是由于大型应用始终无法一次性拿到足够多的 CPU 资源，所以一直无法启动任务，这个大型应用可能就会被“饿死”。

- 资源碎片：由于 wDRF 算法总是追求公平，会把所有剩余资源公平地分配给所有的应用框架，但是这种公平分配可能会导致所有的应用框架在一个分配的周期中都没有足够的资源来运行它的任务。举例来说，假设 Framework A 和 Framework B 都需要 4 个 CPU 来启动任务（单个任务可能仅需要 1 个 CPU，但是这两个应用框架需要一次性同时启动 4 个任务），这时候 Mesos 收到两个 Mesos 节点上各 2 个 CPU 的资源信息，那么 Mesos Master 会基于公平原则，将这些资源以两个资源邀约（每个资源邀约含 2 个 CPU）的形式分别发送给 Framework A 和 Framework B。由于 Framework A 和 Framework B 都需要 4 个 CPU 才能启动任务，那么此时 Framework A 和 Framework B 都会拒绝资源邀约。也就是说，这些资源加在一起能满足一个请求，但基于公平原则，Mesos 不会将多个资源邀约同时发给一个应用框架，导致这个周期内每个应用都无法得到满足。
- “贪婪”：本性“贪婪”或者设计不完善的应用框架可能会在任意情况下都尝试接收所有的资源邀约，即便当时这个应用框架没有任何需要立即执行的任务。在这种情况下，虽然基于公平原则，该应用框架不会占用系统的全部资源，但是仍然会大量地浪费资源。
- 成组调度（gang-scheduling）：是指一些应用框架运行需要的资源类型很多，但与此同时 Mesos 在某个时刻只能提供部分类型的资源，那么这些应用框架可能需要等待比较长的时间，才能等到一个资源邀约里包含所有它需要的资源类型，之后才能开始运行。

针对 wDRF 算法无法解决的问题，Mesos 补充了一系列的资源分配控制手段来进行针对性的处理，下面我们来看一下这些资源分配控制手段。

2. Mesos 资源分配控制手段——资源预留

在现实场景中，某一些应用（如涉及交易结算的应用）需要最高优先级，必须任何时候都能保障其运行。单纯的 wDRF 算法权重也无法 100% 确保无忧：因为可能该应用已经达到，甚至超过了其公平的加权份额，但仍然需要使用更多资源，而此时 Mesos 是不会再向

其发送资源邀约的。那么在这种情况下如何保障应用才能得到绝对的资源数量呢？

资源预留（reservation）是指我们通过提前指定或者实时配置的方式，为指定角色的应用框架在 Mesos 指定节点上预留部分或全部资源，而这些资源只会提供给该角色的应用框架使用。

在介绍资源预留前，我们先了解一下角色的概念，如图 6-18 所示。角色本质上是 Mesos 针对应用框架的资源控制分组。Mesos 对资源的控制范围的划分都是通过应用框架的角色来实现的，因此角色可以为 Mesos 提供多个租户之间的资源隔离；而使用同一个角色来注册的应用框架一般逻辑上是属于同一个租户的。

```
--roles='spark,marathon,kubernetes'
Master启动参数
```

图 6-18 Mesos Master 的启动参数中的 roles 可以配置 Mesos 中允许注册的应用框架角色有哪些

角色的用途如下：

- 权重配置：可以为不同的角色来设置不同的权重，以修改它们之间资源分配的比重。
- 资源的静态预留：在 Mesos Agent 启动时，将部分资源甚至全部资源预留给某个角色，完成资源的静态划分。
- 资源的动态预留：在 Mesos 运行期间，集群的管理员可以通过 HTTP Endpoint 为某些角色动态地预留资源。
- 配额（quota）配置：可以通过为某个角色设置配额来控制这个角色下注册的所有应用框架在整个集群中可使用资源的数量。
- 配置磁盘组：可以为某个角色创建持久卷（persistent volume），隔离不属于这个角色的应用框架，防止其使用这个持久卷。

角色的各种用法在相应章节都会有介绍，这里我们先看看如何通过角色配置 Mesos 资源的静态预留。

一个典型的 Mesos Agent 资源静态预留配置如图 6-19 所示，这个 Mesos Agent 所属节点有 12 个 CPU 和 6 GB 内存，我们通过修改 Mesos Agent 启动参数，将其提供给 Mesos 的资源划分为两部分：一部分是专门提供给角色为 spark 的应用框架使用的资源（8 个 CPU 和 4 GB 内存）；另外一部分是提供给其他默认角色的应用框架使用的资源（4 个 CPU 和 2 GB 内存）。当上述配置经过 Mesos Agent 启动生效之后，Mesos Master 只会将其中的指定资源（如角色 spark 的 8 个 CPU 和 4 GB 内存）发送给指定的应用框架（如只发送给角色

```
mesos-slave --master=<ip>:<port>
--resources=
"cpis:4;mem:2048;cpus(spark):8;
mem(spark):4096"
```

图 6-19 Mesos Agent 启动参数的资源定义中，区分了默认资源和指定角色的资源静态预留

为 spark 的应用框架）。

通过这种方式，我们可以将 Mesos 集群中的资源手动划分为一个个的分区，而每个分区的资源只会提供给相应角色的应用框架使用。

由此可见，资源的静态预留本质就是对 Mesos 节点上的资源做静态分区，而角色就是分区控制的手段。那么静态预留的优点和缺点也就很明显了。

静态预留的优点如下。

- 可以将全部 / 部分资源稳定地预留给一个角色使用，可保证该角色应用资源分配的绝对稳定。
- 可以对不同租户的资源使用实现隔离，互不干扰。

静态预留的缺点如下。

- 如果集群中 Agent 所属节点的资源差异很大，每个节点都需要手动配置静态预留，那么很容易造成配置爆炸，难以管理。
- 静态预留的配置需要重启 Mesos Agent 才能生效，虽然不会影响已经在执行的任务，但会影响重启期间的资源调度，不够灵活。
- 静态预留的资源无法在运行时解除预留 (unreserve，还原为普通资源)，因此预留期间完全无法被其他角色使用，容易造成资源闲置，降低了系统的资源利用率。

可以看到，静态预留的不足大都是由于配置不灵活导致的，所以 Mesos 推出了动态预留。

动态预留（dynamic reservation）与静态预留一样，也是配置 Mesos 节点，为应用框架预留资源，不同的是动态预留的配置方式更加灵活。

我们可以通过以下两种方式配置动态预留：

1）运行时，Mesos 应用框架可以动态地为其在 Mesos 授权列表[⊖]（Mesos ACLs）中授权过的角色预留一些资源。

注意：这里说到的预留可能是为该应用框架自己预留的，也可能是为其他角色的应用框架预留的，也就是说只要配置得当，一个应用框架可以为其他应用框架申请动态预留资源。典型场景如 Marathon、Kubernetes 等应用框架，它们作为应用调度和管理者，需要协调其他应用的运行，因此允许这类应用框架为其管理的其他应用框架申请预留资源是很有意义的。

2）运行时，系统管理人员可以通过 Mesos Master 提供的管理接口 Http Endpoint 直接为某个角色在某个 Mesos Agent 节点上预留资源。

动态预留时，可以为待预留资源添加不同的标签（label），应用框架在拿到预留资源邀约时可以获取这些标签信息，这样应用框架可以轻易地读取出不同预留资源的原始信息（如

⊖ 详细介绍参见 http://mesos.apache.org/documentation/latest/authorization/。

原始预留目的等信息）。

在执行动态预留资源之后，我们可以通过 Master 提供的管理接口 Http Endpoint 的 /slaves 地址查询当前所有 Mesos Agent 的资源预留情况。

除了配置灵活外，动态预留还有另外一个优点，那就是可以动态地解除预留。一种典型的用法是 Kubernetes 作为应用框架向 Mesos Master 发起申请，请求为名为 WebServer 的应用框架所属角色预留资源。Mesos 确认预留资源之后，Kubernetes 启动 WebServer 应用，WebServer 使用该预留资源运行任务。任务执行完毕之后，Kubernetes 向 Mesos Master 发出申请，请求解除之前为 WebServer 预留的资源。而解除预留成功后，该资源被释放出来，可以提供给其他应用框架使用。

更具体的解除预留用法可以参照 Mesos 官网的文档和其他资料，需要注意的是，以下两种情况无法解除预留：

- 预留资源上有正在运行的任务。
- 应用框架使用该预留资源创建了持久卷且未销毁。

3. Mesos 资源分配控制手段——配额

对于资源预留，我们需要注意的是，无论是静态预留还是动态预留，必须指定 Mesos Agent 节点才能进行资源预留，那么这里很容易出现单点故障的问题：假设某应用框架的稳定性和可用性要求非常高，我们为其申请了在某一个 Mesos 集群工作节点上的资源预留，但是当应用框架真正需要运行时，该工作节点发生了宕机，那此时资源预留就随之失效了。

对于资源预留可能存在的单点故障问题，Mesos 引入了配额机制。配额是指：

- 可以在整个集群上为指定角色预留资源，不同于静态预留和动态预留，配额无须指定预留资源所属的 Mesos 节点。
- 如果配额预留的机器出现宕机，Mesos Master 会重新为配额挑选资源预留的机器。
- 配额预留的资源完全不能被其他角色的应用框架使用。

配额具有以下特点。

1）只要集群里有足够的资源就能保证配额生效。

2）只有 Mesos 管理人员才能通过 Mesos Master 配置配额，应用框架不能自行配置配额。

3）配额可以被认为是集群级别的动态预留。

配额的功能非常强大，同时有一些注意事项需要我们了解。

1）目前配额只支持最小保证，不支持最大限制（limits）。

2）目前配额的配置不支持更新，只能先删除再新增。

3）由于配额是集群级别的，所以操作一定要非常谨慎，错误的配额可能会让集群一直

都无法正常工作。

4）目前配额的配置只能作用于标量资源（即支持 CPU、GPU、内存和磁盘，不支持端口资源）。

5）配额的正常运作依赖于 Mesos Master 的资源调度器，因此自定义的 Mesos Master 资源调度器可能导致配额无法正常工作。

配额的工作机制有如下特性。

1）前置检查：配额配置生效前会检查设置，如果某项配额中申请的资源超过了整个集群的资源，那么该配额默认会被自动忽略；如果需要集群扩容，事先知道配额加上已用资源可能会超过集群现有资源的总额，那么我们可以通过添加 force 参数来关闭这个前置检查。

2）为了满足配额申请，Mesos Master 可能会撤销一些已发出但未被应用框架确认的资源邀约，撤销资源邀约有 3 个原则。

- 撤销的肯定是至少有配额申请中那么多资源的资源邀约。
- 如果某个 Mesos Agent 上的一个资源邀约被撤销了，那么该 Mesos Agent 上所有的未被确认的资源邀约都会被撤销，以便于尽可能地满足配额申请。
- 撤销资源邀约的 Mesos Agent 数目大于或等于配额申请中角色的应用框架数目，目的是为了尽可能使得每个应用框架都能拿到资源邀约。

通过静态预留、动态预留和配额这 3 种资源分配控制手段，我们可以分别解决资源分区的问题、实时调整资源分配的问题以及单点故障的问题。

4. Mesos 资源分配控制手段——超售

为了应对峰值负载以及突发性的系统负载抖动，一般的服务器集群都会准备远超平均使用量的资源，而大多数时间里，这些资源都无法得到充分的利用。超售（Oversubscription）可以充分利用已分配出去但未被应用框架实际使用的资源来执行一些优先级较低的任务，如后台分析、视频 / 图像处理、电路仿真等。这种暂时借用的资源可能随时会被撤销，因此也称之为可撤销资源（revocable resource）。当系统压力增大时，可撤销资源可能会被其实际拥有者（应用框架）强制征用，而运行于其上的低优先级任务可能会被强制终止，以便于将该资源及时释放出来。

如图 6-20 所示，Mesos 0.23.0 版本引入了超售机制，在扩展了 Mesos Master 和 Mesos Agent 的同时，还为 Mesos Agent 添加了两个新组件：资源评估器（resource estimator）和服务质量控制（Quality of Service，QoS）。资源评估器用来评估和计算 Mesos Agent 节点上可撤销资源的数量，而 QoS 用来确保可撤销资源不会多到影响正常任务的执行。

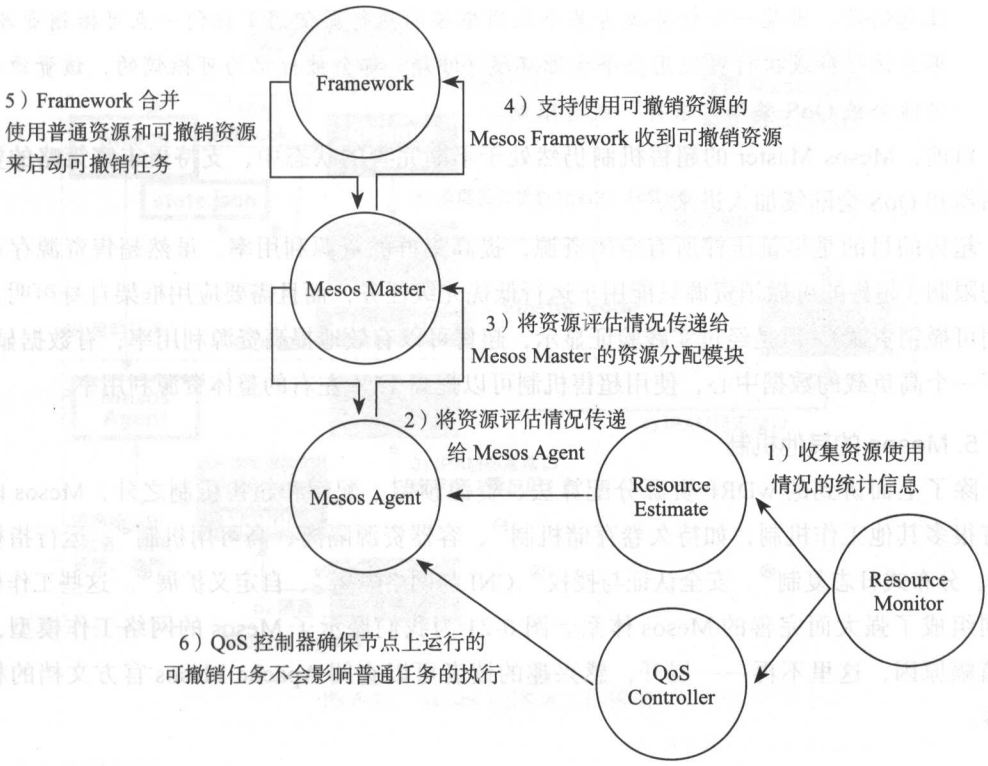


图 6-20 Mesos 资源超售的工作流程

超售机制的工作流程如下。

- 资源评估器持续地读取资源监控器里统计的资源数据，根据自己内部的算法对节点资源数量进行评估，并计算出该节点最新的可撤销资源数量，所有这些信息会被记录在评估结果中。
- 资源评估器将每次的评估结果都发送给该节点上的 Mesos Agent。
- 当 Mesos Agent 发现某一次评估结果与上一次不同时，会将本次评估结果发送给 Mesos Master。
- Mesos Master 收到评估结果后，将评估结果中的可撤销资源以资源邀约的形式发送给那些支持使用可撤销资源运行任务的应用框架（注：应用框架需要在注册到 Mesos Master 上时，显式地声明自己可以使用可撤销资源来执行任务，只有这样该应用框架才会接收到可撤销资源的邀约）。
- 应用框架使用可撤销资源执行任务，这些任务被标记为可撤销任务（revocable task）。
- QoS 持续监控 Agent 节点的资源使用情况，一旦发现可能出现资源紧缺，那么 QoS 就会通知 Agent，Agent 上报 Mesos Master 来“杀掉”或者限制可撤销任务。需要

注意的是：当某一个任务或者某个应用框架的执行器使用了任何一点可撤销资源时，那么该任务或执行器使用整个资源环境（沙箱）都会被标记为可撤销的，该资源环境可能会被 QoS 整体“杀掉”或者限制。

目前，Mesos Master 的超售机制仍然处于不断完善的状态中，支持更丰富策略的资源评估器和 QoS 会陆续加入进来。

超售的目的是尽量压榨所有空闲资源，提高集群的资源利用率。虽然超售资源存在一定的限制（超售的可撤销资源只能用于运行低优先级任务，而且需要应用框架自身声明才能使用可撤销资源），但是经过实践验证显示，超售可以有效地提高资源利用率，有数据显示，对于一个高负载的数据中心，使用超售机制可以提高 15% 左右的整体资源利用率。

5. Mesos 的其他机制

除了上面讲到的 wDRF 资源分配算法、资源预留、配额和超售机制之外，Mesos 内部还有很多其他工作机制，如持久卷存储机制㊀、容器资源隔离、高可用机制㊁、运行指标监控㊂、分布式日志复制㊃、安全认证与授权㊄、CNI 与网络隔离㊅、自定义扩展㊆。这些工作机制共同组成了强大而完善的 Mesos 体系。图 6-21 为我们展示了 Mesos 的网络工作模型，由于篇幅原因，这里不再一一展开，感兴趣的读者可以查阅 Apache Mesos 官方文档的相关内容。

6.2.5 Mesos 的运维和管理

在前面的章节中，我们花大量的篇幅介绍了 Mesos 的工作机制，尤其是详细介绍了 Mesos 对于资源的管理手段，那么在有了理论基础的前提下，我们如何开始 Mesos 实战呢？在本章，我们将介绍 Mesos 的安装部署、运维，以及 Mesos 常用应用框架的相关知识。

1. Mesos 的安装部署

Mesos 有两种安装方式：一是从 Apache Mesos 官网下载源代码编译安装；二是从 Mesosphere 官网下载二进制文件安装。两种安装方式的说明如下。

㊀ 详细介绍参见 http://mesos.apache.org/documentation/latest/persistent-volume/。

㊁ 详细介绍参见 http://mesos.apache.org/documentation/latest/high-availability/。

㊂ 详细介绍参见 http://mesos.apache.org/documentation/latest/monitoring/。

㊃ 详细介绍参见 http://mesos.apache.org/documentation/latest/replicated-log-internals/。

㊄ 详细介绍参见 http://mesos.apache.org/documentation/latest/authentication/ 和 http://mesos.apache.org/documentation/latest/authorization/。

㊅ 详细介绍参见 http://mesos.apache.org/documentation/latest/networking/。

㊆ 详细介绍参见 http://mesos.apache.org/documentation/latest/allocation-module/ 和 http://mesos.apache.org/documentation/latest/modules/。

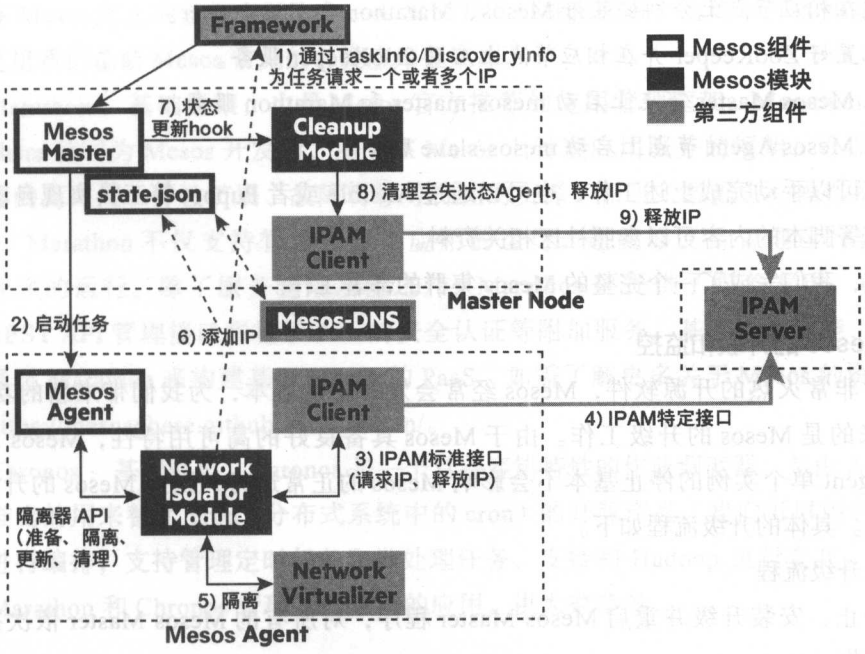


图 6-21 Mesos 的网络工作模型

（1）编译安装

- 下载源码或者使用 git clone Mesos 的 Apache git repository。
- 详细步骤请参照 http://mesos.apache.org/gettingstarted/ 官方文档来执行。

（2）二进制文件安装

- 从 Mesosphere 官网下载 Mesos 源配置文件之后，使用包管理工具执行 Mesos 的安装，源配置文件的下载地址为 https://open.mesosphere.com/downloads/mesos/。
- 详细步骤请参照 https://open.mesosphere.com/downloads/mesos/#installation。

特别需要注意的是：无论是编译安装还是二进制安装都需要连接到互联网才能顺利完成。特别是编译安装，由于编译过程会下载大量依赖文件，因此很难实现完全的离线编译安装。如果目标集群网络无法直接接入互联网，可以考虑通过创建能接入互联网的、与目标集群主机环境相同的虚拟机来直接编译安装 Mesos，再将编译安装好之后的 Mesos 相关文件统一复制到目标集群机器上使用。二进制安装同样存在这个问题，对于 CentOS 或者 RedHat 系统，我们同样可以通过虚拟机配置 Mesos 的 yum 源，使用 yum-utils 的 yum --resolve localinstall 命令下载所有依赖的二进制文件，并复制到目标集群进行统一安装。

通过上面两种方式，我们可以顺利完成 Mesos 的安装工作，与此同时，我们还需要安装和启动其他的组件来配合 Mesos 工作，流程如下。

- 先在相应节点上分别安装好 Mesos、Marathon 和 ZooKeeper。
- 配置好 ZooKeeper 并在相应节点上启动 ZooKeeper 服务。
- 在 Mesos Master 节点上启动 mesos-master 和 Marathon 服务。
- 在 Mesos Agent 节点上启动 mesos-slave 服务。

我们可以手动完成上述工作，也可以通过 Ansible 或者 Puppet 等工具实现自动化部署，自动化部署脚本的内容可以参照社区相关资料。

至此，我们完成了一个完整的 Mesos 集群的部署工作。

2. Mesos 的升级和监控

作为非常火热的开源软件，Mesos 经常会发布新的版本，为我们带来新的功能特性，随之而来的是 Mesos 的升级工作。由于 Mesos 具备良好的高可用特性，Mesos Master 或 Mesos Agent 单个实例的停止基本不会影响 Mesos 的正常运行，因此 Mesos 的升级工作也非常简单。具体的升级流程如下。

（1）升级流程

- 停止、安装升级并重启 Mesos Master 程序，对所有的 Mesos Master 依次执行这项工作。
- 停止、安装升级并重启 Mesos Agent 程序，对所有的 Mesos Agent 执行这项工作（可以并行执行）。
- 如果有必要，升级调度器并链接新的 Mesos 库文件，重启调度器。
- 如果有必要，升级执行器并链接新的 Mesos 库文件，重启执行器。
- 在特殊情况下，如果新版本 Mesos 需要升级操作系统的内核，那么可能需要重启主机（这种情况非常罕见，更新日志中会有特别的强调说明）。

可以看出，整个升级流程非常简单而且很平滑。

（2）Mesos 的运维监控

作为数据中心操作系统的内核，对 Mesos 运行状态的监控非常重要。Mesos 可以与现有监控方案轻松地集成到一起，支持主流的开源监控方案和商业监控方案，并且有插件支持 Nagios、collectd 等监控工具。同时，Mesos HTTP Endpoint 的 /metrics/snapshot 接口提供详细的运行指标报告，可以为定制化开发的监控工具提供丰富的基础数据。

6.3 Mesos Framework

6.3.1 Mesos 常用的 Framework

在数据中心操作系统架构中，Mesos 的角色是系统内核，而各个应用框架（Framework）

是运行在 Mesos 之上的软件。那么 Mesos 集群要对外提供服务，就离不开各种各样的应用框架。这里我们总结 Mesos 常见的各类应用框架如下。

- Marathon：基础框架，也是 Mesos 官方推荐的应用管理框架。Marathon 是 Mesosphere 公司为 Mesos 开发的，用于在 Mesos 上管理应用服务的框架，可以提供对服务的高可用、弹性扩展、故障转移等特性的支撑，相当于数据中心操作系统的 init.d。Marathon 不仅支持管理容器类应用的运行，还能够支持管理运行普通 Linux 程序的运行。除了服务管理功能之外，Marathon 还提供了服务发现、负载均衡、REST API 管理接口和基于 SSL 的安全认证等附加服务。基于这些功能，我们可以通过 Marathon 来构建基于 Mesos 的 PaaS。如需了解更多关于 Mesos 的资料可关注 https://mesosphere.github.io/marathon/。
- Chronos：基础框架。Chronos 是一个具备容错特性的作业调度器，是由 Airbnb 公司推出的用来替代 cron（分布式系统中的 cron）的开源产品，我们可以用它来对作业进行编排，支持管理定时任务和批处理任务，支持和 Hadoop 进行交互。图 6-22 是 Marathon 和 Chronos 管理 Mesos 上的应用，供大家参考。

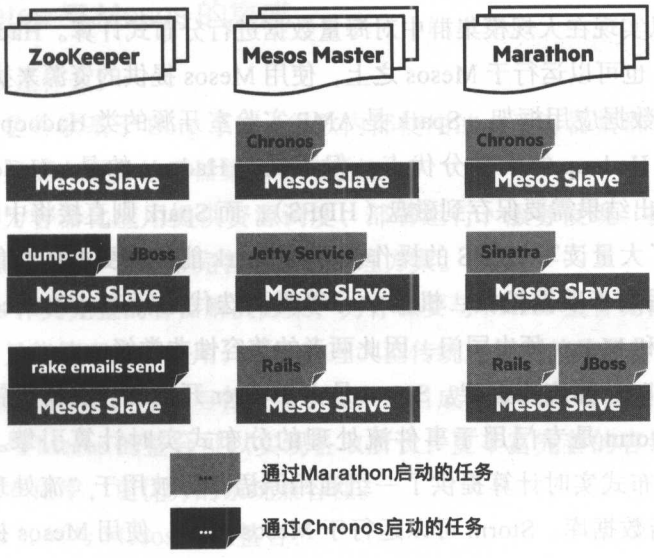


图 6-22 Marathon 和 Chronos 管理 Mesos 上的应用

- Aurora：基础框架。Aurora 是 Apache 的开源项目，是管理长期服务和计划作业的 Mesos 框架。Aurora 的功能类似于 Marathon 和 Chronos 的结合，可同时支持管理长期服务和批处理任务。Aurora 是面向任务的设计，对任务抢占有良好的支持。
- Mesos-DNS：服务发现。如图 6-23 所示，Mesos-DNS 是 Mesos 内置的服务发现框

架，是一个简单易用、无状态、无容错机制的 DNS 服务，无须任何配置即可快速接入 Mesos，为 Mesos 提供简单的服务发现功能。

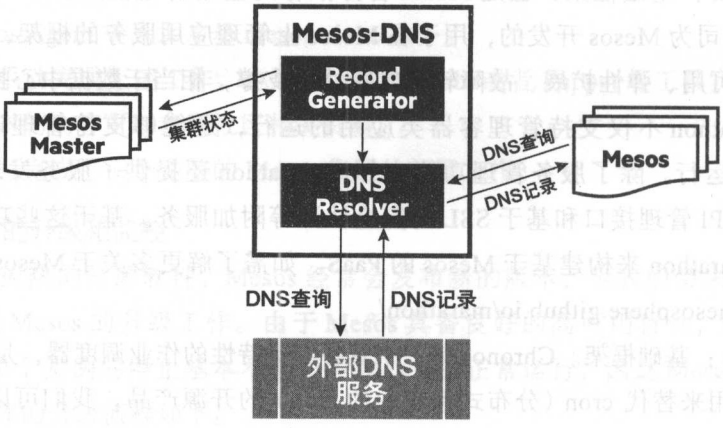


图 6-23 Mesos-DNS 工作模型

- Hadoop：大数据应用框架。Hadoop 是 Apache 基于谷歌大数据论文实现的开源软件框架，可以实现在大规模集群中对海量数据进行分布式计算。Hadoop 有自己的资源管理模块，也可以运行于 Mesos 之上，使用 Mesos 提供的资源来执行任务。
- Spark：大数据应用框架。Spark 是 AMP 实验室开源的类 Hadoop 的通用并行框架。Spark 拥有 Hadoop 的大部分优点，但不同于 Hadoop 的是，Hadoop MapReduce 的 Job 中间输出结果需要保存到磁盘（HDFS）；而 Spark 则直接将中间结果保存在内存中，减少了大量读写 HDFS 的操作。因此 Spark 能提供更好的性能和更快的响应速度，更加适用于数据挖掘、机器学习等需要迭代运行 MapReduce 的场景。特别的是，Spark 和 Mesos 师出同门，因此两者的兼容性非常好。
- Storm：实时计算应用框架。Storm 是由 Twitter 开源的、支持容错的分布式实时计算系统。Storm 是专门用于事件流处理的分布式实时计算引擎。不同于 Hadoop，Storm 为分布式实时计算提供了一组通用原语，可被用于“流处理”之中实时处理消息并更新数据库。Storm 可以运行于 Mesos 之上，使用 Mesos 提供的资源来执行任务。
- Cassandra：分布式数据库。Cassandra 是由 Facebook 开源的分布式 NoSQL 数据库，集 Google BigTable 的数据模型与 Amazon Dynamo 的完全分布式架构于一身。Cassandra 具有良好的可扩展性，被 Instagram、eBay、Netflix 等知名网站所采纳，是一种非常流行的分布式结构化数据存储方案。
- Myriad：集成框架。Myriad 是由 eBay 开源的项目，目前是 Apache 的孵化项目。

Myriad 的目标是实现 Apache Hadoop YARN on Mesos，它将 Mesos 和 YARN 调度器连接起来，使得 Mesos 可以管理 YARN 的资源。当 YARN 中有作业请求资源时，YARN 的资源管理器会先通过 Myriad 的调度器来调度资源，这里 Myriad 的调度就是 Mesos 里应用框架对资源的第二级调度。第二级调度信息会由 Myriad 传递给 Mesos Master，接下来 Mesos Master 会把任务的调度信息发给 Mesos Agent，Mesos Agent 在准备好资源后会调用 Myriad 的执行器，Myriad 执行器的作用是运行 YARN 的节点管理（node manager）器。当 Myriad 在 Mesos 分配的资源上加载 YARN 节点管理器后，YARN 节点管理器就会和 YARN 资源管理器通过通信来确认作业可用的资源，YARN 可以以自己认为适合的方法来使用这些资源。Myriad 为 Mesos 资源池和 YARN 的资源管理之间提供了无缝的桥梁。

- Kubernetes：基础框架，也是提供 PaaS 平台支撑能力的大型综合框架。Kubernetes 可以说是目前人气最高的、最引人注目的容器编排管理技术，具备成为容器编排管理技术事实标准的潜质，因此我们在下一节对 Kubernetes 及其与 Mesos 的集成做单独的介绍。

6.3.2 Kubernetes 与 Mesos 的集成

1）Kubernetes 是什么？

- Kubernetes 是谷歌基于 Borg 系统（谷歌内部使用的大型容器集群管理系统）实现的开源项目，是面向服务的容器集群管理系统。
- Kubernetes 为容器化应用提供资源调度、部署运行、服务发现、扩容缩容、故障恢复等特性支持，是一套非常完善的容器管理方案。

2）Kubernetes 作为完整的容器解决方案，为什么要与 Mesos 整合呢？

- Kubernetes 只能管理容器应用，无法处理大型传统应用和其他非容器类应用。
- Mesos 擅长资源调度，可以与各种需求复杂或自成体系的应用框架集成。
- Kubernetes 与 Mesos 的整合可以实现各取所长：更丰富完善的容器管理策略，更高的集群资源利用率，更优秀的系统兼容性。

3）Kubernetes 如何与 Mesos 实现整合？

- Kubernetes 官方项目已经合并了 Kubernetes on Mesos 的源代码。
- 在 Kubernetes 项目编译前，配置好环境变量 KUBERNETES_CONTRIB=mesos，再重新编译 Kubernetes 即可。
- 详细的集成过程可以参见官方文档：https://kubernetes.io/docs/getting-started-guides/mesos/。

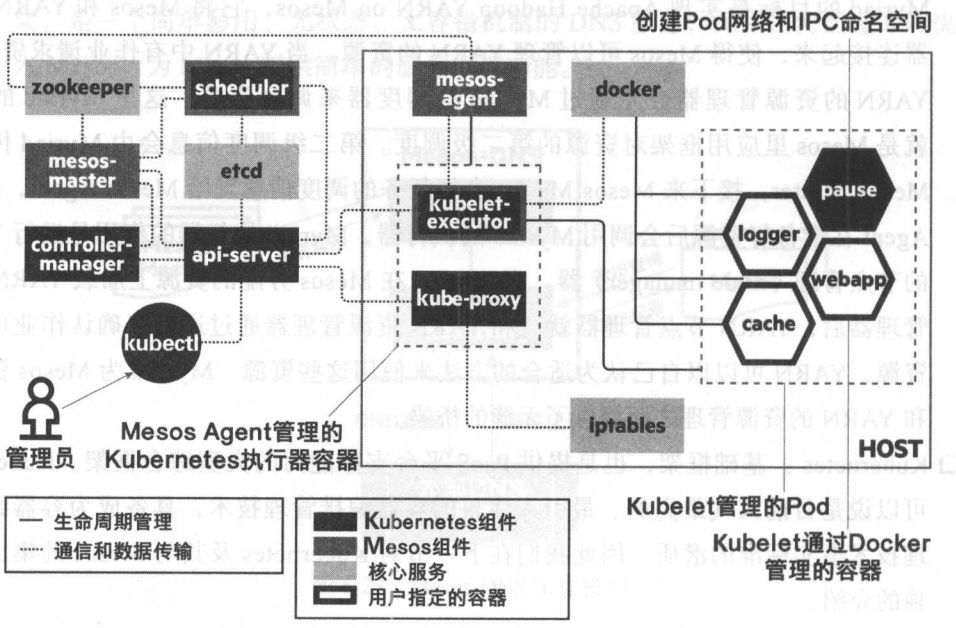


图 6-24 Kubernetes on Mesos 架构

Kubernetes on Mesos 架构如图 6-24 所示，其工作流程如下。

- Mesos 作为资源管理中心，将集群资源邀约发送给 Kubernetes 在 Mesos 中的调度器。这个调度器的职能是由 Kubernetes 自己的 api-server、scheduler 和 controller-manager 共同承担的，api-server 负责与 Mesos Master 的直接通信，scheduler 负责 Kubernetes 内部任务的调度，而 controller-manager 负责 Kubernetes 内部任务的执行控制。
- Kubernetes on Mesos 的调度器将调度信息发送给 Mesos Master。
- Mesos Master 将任务信息下发给 Mesos Agent。
- Mesos Agent 根据任务信息创建资源环境，并根据任务信息中的 Kubernetes 调度信息，来调用 Kubernetes 在 Mesos 中的执行器启动相应的任务。这个执行器的职能是由 Kubernetes 运行于 Mesos Agent 工作节点上的 kubelet 和 kube-proxy 共同承担的，kubelet 负责管理 Kubernetes 任务的生命周期，而 kube-proxy 则管理 Kubernetes 服务相关的节点网络配置。

Kubernetes on Mesos 只是将原始 Kubernetes 的资源调度模式改变为 Mesos-Kubernetes 的二级资源调度，而 Kubernetes 和 Mesos 的功能和特性几乎得到了完整的保留，因此能达到相辅相成、相得益彰的效果。

6.4 Mesos 发展远景分析

6.4.1 Mesos 的技术特点

1. Mesos 与 Kubernetes 的比较

Mesos 与 Kubernetes 的技术对比如图 6-25 所示，两者的关系如下。

- 目标不同：Mesos 以资源调度为核心；Kubernetes 是面向服务的设计。
- 定位不同：Mesos 定位于资源的统一管理，以灵活的两级调度方式支持各种复杂应用场景；Kubernetes 则侧重于容器的编排管理，为容器化应用提供强大的管理方案。
- 优势不同：Mesos 常用于协同差异化的各类应用框架共同运行，但本身对于容器的管理较为薄弱，必须依靠其他基础框架来实现；Kubernetes 对容器化应用的管理得心应手，但对底层资源的复杂管理需求以及非容器类应用的管理比较乏力。
- 可以兼得：Kubernetes on Mesos 可以让两者各取所长，发挥出更大的威力。

应用	Mesos	Kubernetes
实现语言	C++	Go
资源分配	wDRF算法可能出现资源"饿死"	Policy Chain过滤和评分
对Hadoop的支持	粗粒度，整个Hadoop运行	不支持
对Docker的支持	好	非常好
社区参与者	较多	非常多
对长服务的支持	好	好
两者关系	Mesos 以资源调度为核心；Kubernetes 是面向服务的设计。 Mesos 更多定位在资源的抽象和管理上，以便支持各种应用； Kubernetes 则侧重于容器的编排管理，为容器化时代提供一套强大的方案 Mesos 本身对于容器的管理，以及以容器为载体的服务管理非常乏力，必须依靠类似于 Marathon 之类的框架实现；Kubernetes 则对底层资源管理及非容器的应用无能为力 Kubernetes on Mesos 的集成可以让两者发挥各自的长处，发挥出更大的威力	

图 6-25 Mesos 与 Kubernetes 的技术比较

2. Mesos 与 YARN 的比较

Mesos 与 YARN 的技术对比如图 6-26 所示，两者的关系如下。

- 目标类似：Mesos 和 YARN 都以资源管理为核心。
- 定位不同：Mesos 定位于资源的统一管理，可以支持各种复杂的场景；YARN 虽然被设计为通用资源调度管理平台，但就实际使用情况来看，还是更适合于作为大数据平台的资源管理系统，YARN 对于大数据应用的管理粒度要优于 Mesos。

- 优势不同：Mesos 常用于协同差异化的各类应用框架共同运行，更适合异质资源需求的应用接入，而对大数据应用的管理粒度较粗；YARN 长于大数据应用的管理，但是对非大数据类应用并不擅长。
- 可以兼得：上文介绍的 Myriad 可以实现 YARN on Mesos 的集成。

应用	Mesos	YARN
实现语言	C++	Java
资源分配	wDRF算法可能出现资源“饿死”	资源预留，利用率低
对Hadoop的支持	粗粒度，整个Hadoop运行	细粒度，每个MR为一个应用
对Docker的支持	好	很差
社区参与者	较多	较多
对长服务的支持	好	一般
两者关系	都以资源调度为核心 YARN 借鉴了 DRF 算法；Mesos 更重视对框架的支持。 YARN 更多应用在大数据平台上，对上层计算框架支持得非常好；Mesos 更多定位在资源的抽象和管理上，以便支持各种应用，而不仅仅是计算框架 YARN 更适合管理大数据应用，Mesos 更适合异质资源需求的应用接入 两者的设计目标和功能有很多重叠，但非水火不容：Apache 孵化项目 Myriad 实现了 YARN on Mesos，使得 YARN 调度器可以使用 Mesos 管理的资源，各取所长，相得益彰。	

图 6-26 Mesos 与 YARN 的技术比较

3. Mesos 元框架的选择

如前文所述，Mesos 中有多个可以选择的元框架，我们总结了各类元框架的适用场景，结论见图 6-27，说明如下。

元框架	适用场景
Docker Swarm on Mesos	习惯使用类似于Docker的API 实验性功能验证 Docker粉丝
Marathon on Mesos	使用Mesos来运行长期服务（不局限于容器） 使用Mesos Attributes来控制调度 完善的监控，健康检查和故障恢复 整合HAProxy等应用来实现服务发现 完整的Web UI和REST API，自定义Mesos Framework的管理
Chronos on Mesos	使用Mesos来运行批处理任务（不局限于容器） 定时任务支持（Cron） 支持任务依赖的DAG工作流 完整的Web UI和REST API，自定义Mesos Framework的管理
Kubernetes on Mesos	面对服务的设计理念 强大的Pods特性 基于Label的服务发现，负载均衡和RC控制 其他种种强大特性 急速上升的发展趋势，火热的开源社区，Google强大的后盾

图 6-27 Mesos 常用的元框架适用场景说明

Docker Swarm 目前发展很快，而且有 Docker 这个最主流的容器技术作为基础，相信会有很好的前景；但是 Docker Swarm 目前还相当不完善，功能缺失，稳定性也急需提高，所以如果需要在生产环境中使用 Docker Swarm on Mesos 一定要慎之又慎。

Marathon 是 Mesosphere 公司自己的产品，也是其主推的 Mesos 基础框架，稳定性跟成熟度都毋庸置疑。由于设计理念的差异，Marathon 对于容器化环境下服务管理的灵活性和便利性比 Kubernetes 要稍逊一筹，但仍然不失为一个优秀而全能的框架，是一个可靠的选择。

Chronos 一般是作为 Marathon 的补充出现的。由于 Marathon 不擅长处理批处理任务，所以当我们使用 Marathon 而又需要管理批处理任务时，Chronos 就成了最佳选择。

Kubernetes 可以说是容器调度管理技术中最火热的“明星”。当我们的应用场景面对完全容器化的环境时，Kubernetes 是无可争议的最佳选择。

4. Mesos 的适用场景

基础架构的选择对于一个系统来说至关重要，那么什么情况下我们可以考虑使用 Mesos 作为系统的基础架构呢？如图 6-28 所示，当我们需要实现基础设施虚拟化时，最好的选择是虚拟化技术，如开源的 OpenStack 或者 VMware vSphere 等商业化产品；当我们面对完全容器化的环境时，使用 Kubernetes 或者 Docker Swarm 会来得更加直接；如果系统只运行大数据应用，那么 YARN 作为基础平台是最佳选择；如果我们需要实现一个大型集群的数据中心操作系统，需要同时支撑大数据应用、容器应用和非容器传统应用的运行，还希望能够有效地提高集群的资源利用率，那么 Mesos 可以说是必然的选择。Mesos 开放而灵活的设计，使得它可以兼容各类应用，满足这些复杂的场景需求。

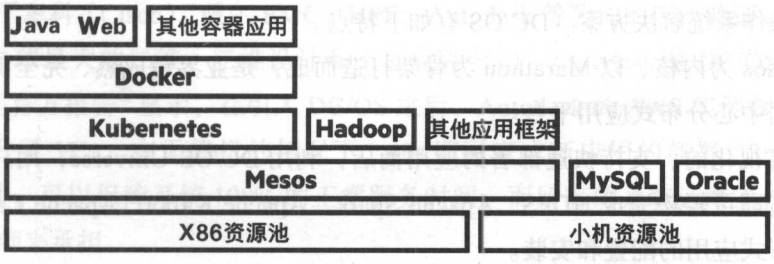


图 6-28 一个使用 Mesos 的数据中心架构案例

在本书里我们介绍了 Mesos 的诸多优点，但同时我们应该认识到每种技术都有其局限性，Mesos 也不例外。了解技术的局限性有利于我们更深刻地理解技术的边界，对我们架构设计和工程实践很有帮助。那么我们现在来看一下 Mesos 有哪些局限性。

- 系统功能过于依赖应用框架。在 Mesos 中，复杂的功能全部过渡给了应用框架，如资源的细粒度调度、应用服务的管理功能等。正因为如此，Mesos 体系中应用服务

的质量基本上完全取决于应用框架的质量。对于一些成熟可靠的应用框架，可以“开箱即食”，但是一旦遇到复杂的需求无法被现有框架满足时，必须自己动手开发应用框架。应用框架的开发上手非常简单，但是如果要实现一个完善高效的应用框架，并不是一件非常简单的工作，学习成本、时间成本和人力成本都不可小觑。

- Mesos 中的应用框架有过高的自由度。由于 Mesos 只限定了应用框架必须实现的接口，对于应用框架的资源申请（不超过公平额度的资源申请）和使用并没有任何限制，因此可能会出现存在设计缺陷或恶意的第三方应用框架滥用系统资源的情况。在使用第三方应用框架时，必须小心谨慎，经过反复测试确认没有问题才能考虑上线使用。
- 对应用服务的管理乏力。由于 Mesos 并不支持直接管理应用服务，所以对于应用服务层最重要的特性（如服务发现、负载均衡、扩容缩容等功能），Mesos 必须通过 Marathon、Kubernetes 等外部应用框架来实现。

上述问题的根源在于 Mesos 的功能过于单一，即只专注于数据中心的资源管理。单独的 Mesos 无法组成一个完整的数据中心操作系统，需要配合各种应用框架才能组成一个完善的方案。与此同时，第三方应用框架良莠不齐，如何选择应用框架并合理地搭配使用就成了一件比较困难的工作。针对这种情况，Mesosphere 公司及时地推出了 DC/OS 平台，将 Mesos、常用应用框架及其他功能组件进行了良好的封装，并提供一系列强大的特性，组成完整的数据中心操作系统解决方案。在本章的最后，我们简要介绍一下 DC/OS。

6.4.2 DC/OS 简介

DC/OS（开源版）是 Mesosphere 公司于 2016 年 4 月推出的开源软件，是完整的端到端的数据中心操作系统解决方案。DC/OS 有如下特点。

- 以 Mesos 为内核、以 Marathon 为骨架打造而成，是业界最成熟、完全满足生产要求的数据中心分布式应用平台。
- 提供可视化的、支持快速部署的应用商店：利用 DC/OS Universe，用户只需通过网页点击即可轻松完成 HDFS、Apache Spark、Apache Kafka、Apache Cassandra 等复杂分布式应用的配置和安装。
- 强大的应用管理能力：DC/OS 内置组件支持应用高可用、服务发现、负载均衡、快速扩展、容灾、故障恢复等功能，同时支持可视化的应用配置和管理。
- 完善的安全机制和监控功能：安全组件贯穿整个 DC/OS 架构，DC/OS 提供基于 GUI 的监控与管理方案，能同时管理数据中心中所有的节点、应用框架以及 DC/OS 内部组件的健康指标，使得运维工作变得非常简单。
- 良好的兼容性：DC/OS 可以运行在裸机、虚拟机或云平台上，能够无视环境差异，为用户提供一致化的体验。

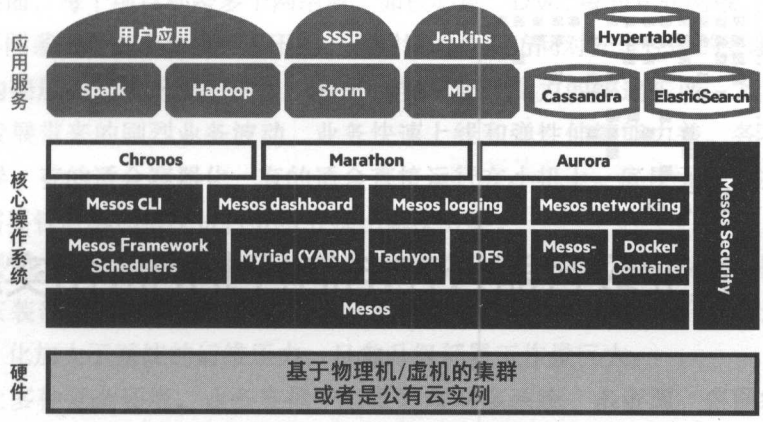


图 6-29 DC/OS 技术架构图

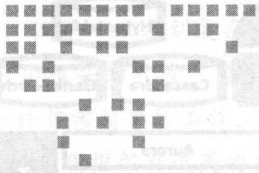
DC/OS 的技术架构（如图 6-29 所示）：底层是资源层，支持裸机、虚拟机和云环境；Mesos 是核心，支撑着上面的各种应用框架；DC/OS 默认安装 Marathon 和 Chronos 作为元框架，用户可以通过应用商店一键安装其他应用框架；此外，DC/OS 内置了监控管理、统一命令行终端、日志、网络管理、安全等功能模块。更多关于 DC/OS 的资料，请参见 https://dcos.io。

不同于 Mesos、Docker Swarm 和 Kubernetes，DC/OS 贯穿基础设施层和应用服务层，是一个功能完整的数据中心操作系统。

开源版 DC/OS 起源于 Mesosphere 公司 2014 年推出的商业版 DC/OS，而商业版 DC/OS 已经得到了思科（Cisco）、戴尔 EMC、HPE、Autodesk 等公司的广泛验证。以 Autodesk 为例，作为全球最大的二维、三维设计和工程软件公司，Autodesk 是 DC/OS 的第一个商业客户。根据官方报告[⊖]显示，在引入 DC/OS 之后，Autodesk 的 AWS 实例数降低了 66%，成本改善提升了 57%，在无停机的情况下，40s 即可完成新应用的部署，3min 即可完成新分区的创建，可以保障系统 100% 的正常服务时间，而所有的系统运维工作只需要 1 名 DevOps 工程师来承担。

可以预见，作为当前开源数据中心操作系统的最佳实现，DC/OS 会被越来越多的组织和个人所接受和认可。我们也希望各位开源软件的爱好者积极参与到 Mesos 和 DC/OS 的开源工作中来，为云计算的发展贡献出自己的力量。

⊖ 详细介绍参见 http://cloudengineering.autodesk.com/blog/2016/04/autodesk-is-forging-ahead-with-dcos.html。



Chapter 7 第 7 章

企业级容器云在电信行业的应用实践

7.1 企业为什么要建设容器云 PaaS 平台

7.1.1 背景

在之前的章节中，我们介绍了构建企业级容器云 PaaS 平台的各关键技术组件，然而，如何运用这些技术来满足用户的业务需求，并不断在实践中解决用户面临的实际问题，为用户带来切实的收益，才能彰显技术本身的神奇威力，验证它的适用性。在企业级容器云的实践案例中，我们基于通信行业的某电信运营商的一级业务支撑系统，进行了企业级容器云 PaaS 平台的设计、开发和应用实践。

某电信运营商一级业务支撑系统是整个企业核心业务集中管理和一点对外的门户，包括多个业务系统，业务模式涵盖了交易、计费、服务等各种电信核心业务模式，系统功能各异、复杂度高，各系统呈烟囱化分散建设。在小型机时代由于主机集成度高、性能稳定、数量较少，多项目集群的建设、运维尚能保持平稳，但随着系统 X86 化逐步推进，多项目集群中分别管理的主机、网络、存储等资源数量成几何级数增长，对项目建设、开发、运维等各个流程都带来颠覆性挑战。

在资源层面，各个项目的资源分散在各个数据中心和私有云里，不能统一管理，每个项目都有独立的资源，分布在不同的机房，特别是部分核心系统分布在 31 个省，对跨数据中心的资源和系统统一管理难度较大。

在网络层面，每个项目都跨多个网络域，如核心域、DMZ 域和互联网域，之间有多层防火墙隔离，不同系统的安全等级要求不同，不同域的数据访问对安全控制的要求也各不相同。

在技术构架层面，缺乏统一的技术和平台实现业务能力的快速扩展，尤其是无法满足互联网业务发展带来的剧烈业务波动，业务快速上线和弹性伸缩能力差。各项目采用的技术和框架各异，有的适合容器化，有的适合直接运行在小机上。底层无法直接进行技术复用和共享，需要针对具体情况进行分析并提出解决方案。

在运维层面，开发环境管理复杂，开发、测试、生产环境无法进行有效隔离，无法实现自动化的安装部署和应用维护，业务的环境、配置依赖问题常常会给系统迁移带来很大的麻烦。X86 化加大了系统的运维压力，日常升级部署工作量巨大。

鉴于以上多种复杂环境，业务支撑中心要实现对多系统、多资源、多网络、多数据中心、多技术架构的统一管理，必须构建企业级容器云 PaaS 平台，才能实现对多项目下多集群的统一管理和集中运维。

7.1.2　试点系统选择

某电信业务支撑系统作为整个一级业务支撑系统的核心系统，是某电信运营商内外部集信息传输交换、服务管控、数据处理、业务支撑、运营开放为一体的综合信息交换枢纽，是连接某电信运营商总部、31 个省公司、各一级业务平台、服务公司、合作伙伴等内外部各应用系统，并对外提供服务的桥梁，是某电信运营商的企业数字神经网络。图 7-1 是某电信业务支撑系统全国部署图，目前承载 200 多个平台的接入，支撑业务包含金融、客服、业务订购、互联网等，峰值业务量达到 10 亿笔 / 天。

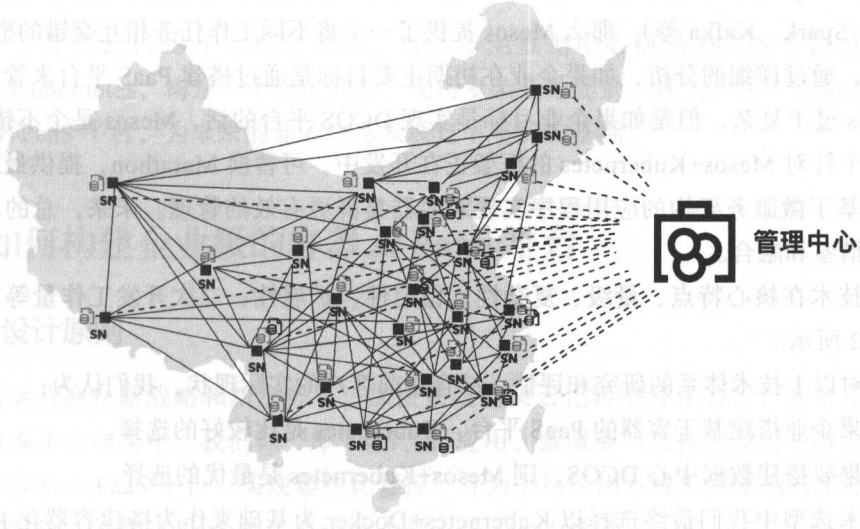


图 7-1　某电信业务支撑系统全国部署图

该系统承载业务具有容量大、实时性强、波动剧烈、增长迅速、重要性强、客户影响大、无状态业务居多等特点，非常适合做 PaaS 平台的试点。因此，我们选择该业务支撑系统作为搭建企业级容器云 PaaS 平台的基础平台。

7.1.3 容器云 PaaS 平台技术选型

在众多的技术框架中，选择何种技术集合作为构建容器云 PaaS 平台的架构框架是非常关键的环节，在某种程度上决定整个项目的成败。目前适用于容器集群管理和大规模部署的，并且得到大规模生产验证的开源产品有 Kubernetes 和 Apache Mesos，这两个平台各有特点。

Kubernetes 是一个全新的基于容器技术的分布式架构的集群管理解决方案，Kubernetes 具有完备的集群管理能力，包括多层次的安全防护和准入机制、多租户应用支撑能力、透明的服务注册和服务发现机制、内建智能负载均衡器、强大的故障发现和自我修复能力、服务滚动升级和在线扩容能力、可扩展的资源自动调度机制，以及多粒度的资源配额管理能力。

利用在容器技术上的实践经验和技术积累，目前 Kubernetes 生态环境热度很高，发展很快。谷歌使用的 Kubernetes 目前每秒会启动大约 7000 个容器，每周可能会超过 20 亿个容器。

Mesos 目前作为 DCOS（Data Center Operation System）理念的实现者，也得到了很多企业的关注。但是 Mesos 如果作为容器集群的管理者，需要通过 Marathon 框架支撑，另外还需要单独增加很多 Kubernetes 内置的功能，如 Proxy、Service DNS，以及集群的动态伸缩要求和 Proxy 负载策略的数据同步、应用监控等。如果你拥有已经存在的多个工作任务（Hadoop、Spark、Kafka 等），那么 Mesos 提供了一个将不同工作任务相互交错的框架。

因此，通过详细的分析，如果企业在初期主要目标是通过搭建 PaaS 平台来管理容器集群，Mesos 过于复杂，但是如果企业目标是实现 DCOS 平台的话，Mesos 是个不错的选择。另外，一个针对 Mesos+Kubernetes 的框架正在开发中，可替换 Marathon，提供最理想的方式以构建基于微服务架构的应用程序实现对容器集群更有效的管理。未来，总的趋势是两者不断地借鉴和融合。

相关技术在核心特点、量级、复杂性、稳定性、扩展性、二次开发工作量等方面的比较如图 7-2 所示。

通过对以上技术体系的研究和评估，结合目前客户的实际现状，我们认为：

- 如果企业搭建基于容器的 PaaS 平台，Kubernetes 是比较好的选择。
- 如果要搭建数据中心 DCOS，则 Mesos+Kubernetes 是最优的选择。

在技术选型中我们最终选择以 Kubernetes+Docker 为基础来作为搭建容器化 PaaS 平台

的方案。其优点是已经经过谷歌十多年的生产验证，成熟度高，支持裸机、VM 等混合部署，适合多种应用场景，Kubernetes 可以用最快的、最简单的、最轻量级的方式来解决目前企业 IT 建设存在的问题，并有助于进行面向集群的开发。而且很多厂商已经开始支持 Kubernetes，例如微软、IBM、Red Hat、CoreOS、MesoSphere、VMware 等。社区的热度很高，功能也在快速地增强中。

	Kubernetes	Mesos
定位	• 专注于Docker容器的集群管理 • 以service的角度定义容器的应用，产生微服务 • 整个框架考虑了很多生产中需要的功能，比如Proxy、Service DNS、Liveness Probe等，基本不用二次开发就能应用到生产	Mesos是一个分布式系统内核，组织不同类型的主机放在一起当一台逻辑计算电脑。它专注资源的管理和任务调度，并不是针对容器管理的。Mesos上所有的应用部署都需要有专门的框架支撑，如支撑Docker必须安装Marathon。安装Spark和Hadoop都需要不同的框架
对容器支撑	天生就是针对容器和应用的云化，通过微服务的理念对容器进行服务化包装	支撑Docker必须安装Marathon框架。Mesos只关注应用层资源的管理，其余由框架完成
对资源的控制	Kubernetes本身具备资源管控能力，可以控制容器对资源的调用	Mesos对所有的主机虚拟成一个大的CPU，内存池，可以定义资源分配，也可以动态调配
是否可以分区	Kubernetes能通过namespace和node进行集群分区，能控制主机、CPU和内存	可以，可以定义到CPU、内存、磁盘等
开发成本	原生集成了Service Proxy、Service DNS，集群动态扩展对Proxy的实时更新，基本没有二次开发成本，而且便于多集群的集成	要实现生产应用需要增加很多功能，如HA Proxy、Service DNS等，需要自己实现集群扩展和Proxy的集成，二次开发成本高，需要专业的实施团队
非docker应用的集成	对于不能实现Docker化的应用，通过外部Service方式集成进集群	必须自行开发Framework来集成到Mesos里面

图 7-2　Kubernetes 与 Mesos 比较示意图

在企业容器云 PaaS 平台稳定之后，再逐步向数据中心级的 DCOS 平台过渡，整合 Mesos 和 Kubernetes，构建一个稳定性强、支持复杂业务场景、强大弹性扩展能力的电信行业 DCOS+PaaS 平台，为未来的业务快速发展打下坚实的基础。

7.2　如何构建企业级的容器云 PaaS 平台

7.2.1　设计原则

电信运营商在新战略和新形势下，加快系统构架云化转型势在必行，业务支撑系统正在由分散逐步走向集中。我们以“厚 PaaS、轻应用、微服务”为构架设计总体原则，来构建企业级容器云 PaaS 平台，实现集中化管控，并为上层应用系统提供基础技术服务和公共业务组件服务。容器云 PaaS 平台架构设计遵从以下设计原则。

（1）先进性和成熟性

互联网技术发展迅速，新的技术不断涌现并趋于成熟，系统设计需在满足实用性的基础上，立足高起点，选用先进性和成熟性融合较好的技术，既能确保平台领先，满足 3 ~ 5 年的技术发展需要，也要保证经过较好的实践验证。

（2）开放性与标准化

平台选用的所有产品和技术都需要符合国际、国家相关标准，是开放的可兼容系统，并能与不同厂商的产品兼容，以有效地保护投资。因此在总体设计中应秉承开放式、松耦合、标准化设计原则，使系统有适应外界环境变化的能力，易于调整、扩充和组合，最大限度满足业务要求。

（3）可靠性与安全性原则

安全可靠的运行是整个系统建设的基础。提供良好的安全可靠性策略，支持多种安全可靠性技术手段，制定严格的安全可靠性管理措施。系统要具备容错、备份及自诊断模块，便于快速判断故障点并排除。要配置严密的数据安全体系，避免非法入侵，确保系统数据的准确性、正确性，防止异常情况的发生。

（4）可维护性和易用性

提供界面化的管理工具，实现对系统的日常维护，同时提供及时可靠的告警和通知机制，实现对问题和故障的预警和及时通知。

（5）运营流程的敏捷性与规范性原则

系统具备开发更新的敏捷性，能够敏捷地支撑各类复杂电信业务的交付，同时流程和服务复杂的应用也需控制，具备可管理性。因此，平台具备一个高效的流程设计和管理工具就成为一个必要条件，以便未来的业务变更开发过程能够规范、高效、快捷。

7.2.2 容器云 PaaS 平台总体规划和建设路径

某电信运营商企业级容器云 PaaS 平台的建设具有非常重要的战略意义，是整个企业基础设施平台云化建设的标杆项目，因此，必须具有一个长远性、全局性、方向性的整体规划。好的规划就像一座桥，可以将我们现在所处的位置和想要达到的目标联结起来，可以使我们的行动目标更加清晰，行动方案更具可行性和可控性。

在容器云 PaaS 平台的前期规划中，我们对实现容器云 PaaS 平台提出分阶段实施的目标规划，如图 7-3 所示。

（1）第一阶段：容器云 PaaS 基础平台搭建

- ❑ 架构设计、集群设计、资源分区。
- ❑ 搭建容器云 PaaS 基础平台和容器云管理平台，建立服务控制中心。
- ❑ 创建租户，通过租户对各项目进行统一管理。

❑ 打通容器云 PaaS 平台与 IaaS 层资源（主机、网络、存储）的统一管理接口。

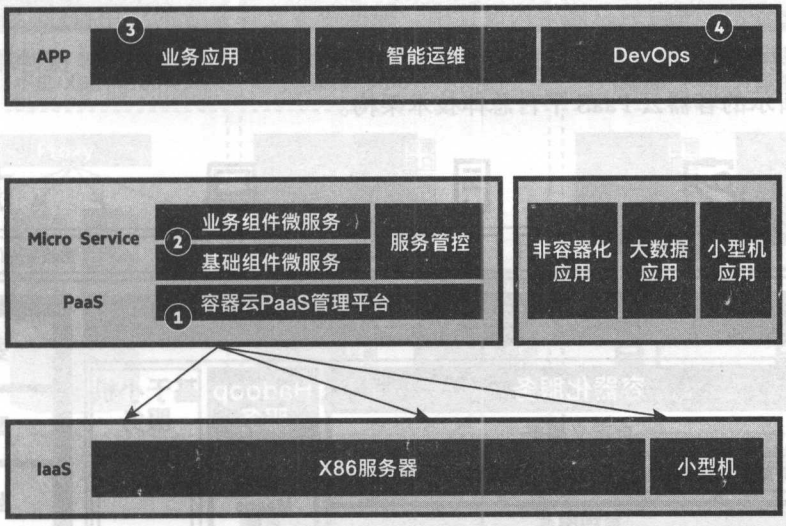


图 7-3　容器云 PaaS 平台总体规划和建设路径

（2）第二阶段：微服务化和容器化改造

❑ 平台基础组件（MySQL、GlusterFS、Memcache、Redis 等）的微服务化和容器化设计、改造、迁移和部署。

❑ 平台业务组件的微服务化和容器化设计、改造、迁移和部署。

❑ 将非容器化应用纳入统一的管理框架，实现资源层面的共享和调度。

（3）第三阶段：应用项目分批迁移

❑ 应用项目的分批、分阶段迁移和部署，分析并选取部分应用项目，迁移到容器云 PaaS 平台。

❑ 验证应用在容器云平台运行稳定性，积累系统迁移经验。

❑ 建立标准化的镜像打包规范、应用开发规范、应用运营管理规范、运维手册等。

（4）第四阶段：统一管理，持续运营

❑ 将微服务设计、开发、构建、部署、上线等过程与 DevOps 管理流程和工具结合起来，实现持续集成和交付流水线，建立 DevOps 过程管理规范。

❑ 建立基于机器学习的智能运维，实现自动巡检，日常任务自动化，实现业务趋势预测和运维助手机器人。

通过对以上四个阶段的分步实施、效果评估和持续改进，为某电信运营商企业级容器云 PaaS 平台的落地提供一个快速可行路径。

7.2.3 容器云 PaaS 平台总体技术架构

在容器云 PaaS 平台的总体架构设计中，根据业务支撑系统的应用特性、部署现状、未来发展和云化实践的情况，项目技术团队对系统构架做了大量的分析、探讨和预研，提出了如图 7-4 所示的容器云 PaaS 平台总体技术架构。

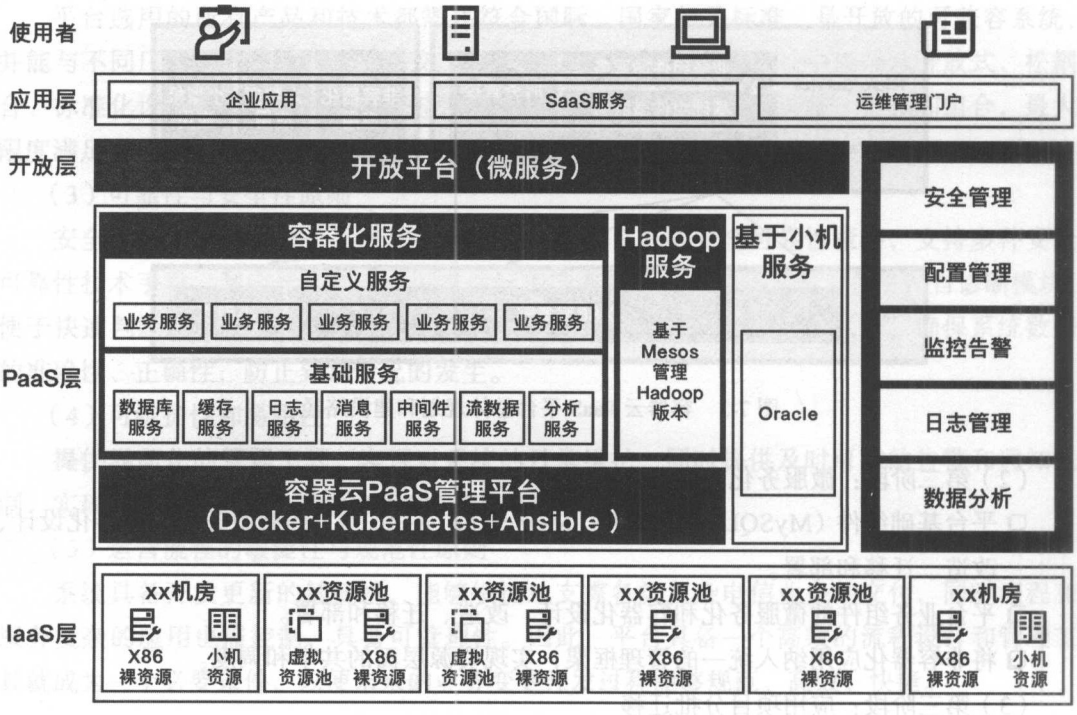


图 7-4 某电信业务支撑系统容器云 PaaS 平台总体技术架构

该架构方案以 Docker 容器为承载，微服务架构为支撑，结合 Kubernetes 作为集群管理平台。既满足了适用于容器化应用的服务统一管理，又满足了不适于容器化的 Hadoop 等服务和基于小机类的应用服务的统一访问和统一管理，既适合当前业务支撑系统发展的实际需求，又具有支撑未来业务高速发展的技术先进性。

本技术架构规划以业务支撑系统典型业务为实践范例，对业务类型和容器模型进行抽象，尽可能考虑其通用性和普适性，归类出以下几种常用的应用容器模型，如图 7-5 所示。

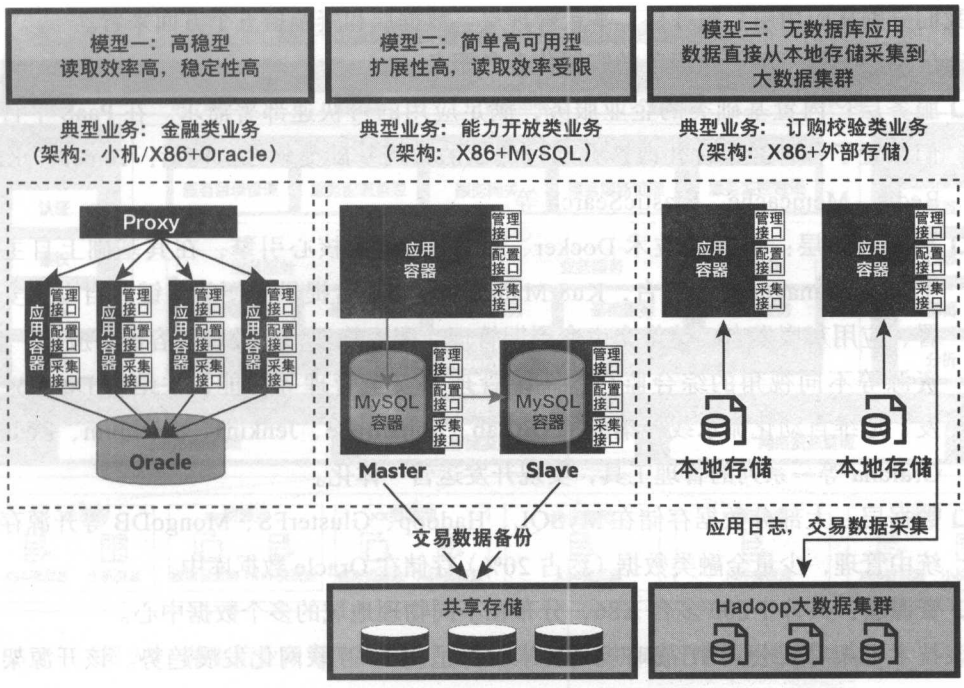


图 7-5　某电信业务支撑系统容器设计模型

7.2.4　容器云 PaaS 平台采用的开源技术框架

企业级容器云 PaaS 平台的技术架构遵循开源成熟标准和框架来实现，如图 7-6 所示。

图 7-6　某电信业务支撑系统容器云 PaaS 平台采用的开源化技术框架

我们分别从应用层、服务层、平台管理层、数据层和资源层五个方面来看。

- 应用层：开发基于微服务的容器化业务服务。
- 服务层：内置基础类的企业服务，满足应用的可快速部署需求。在 PaaS 平台中我们预置了各种经过优化改造并通过测试的常用基础类软件，包括：Tomcat、Kafka、Redis、Memcache、ElasticSearch 等。
- 平台管理层：以开源技术 Docker、Kubernetes 为核心引擎，在其基础上自主开发了 Ku8 Manager 管理平台，Ku8 Manager 管理平台提供简便的一键式自动化安装部署、应用灰度发布、镜像全生命周期管理、配置管理，以及基于容器、应用、服务、资源等不同视角的综合监控、系统管理和安全管理等功能，并结合 DevOps 的开发、运维自动化流水线，集成了 GitLab、Sonarqube、Jenkins、Selenium、Redmine、Grafana 等一系列的管理工具，实现开发运营一体化。
- 数据层：大部分数据存储在 MySQL、Hadoop、GlusterFS、MongoDB 等开源存储系统中管理，少量金融类数据（约占 20%）存储在 Oracle 数据库中。
- 资源层：共计 1 200 多台 X86，分布在不同物理地域的多个数据中心。

该技术框架符合公司 IT 战略规划要求，并适应 IT 互联网化发展趋势。该开源架构在大并发业务量下的稳定性和吞吐性能已通过生产验证，各方面性能（可靠性、扩展性、安全性、可维护性等）比旧有的商业体系架构表现得更加优秀。

7.2.5 基于微服务的容器化 PaaS 平台应用管理架构

在系统架构的自然演进下，产生了微服务，微服务更强调业务服务的自治性及原子性。Docker 容器化技术的出现，有效解决了微服务的环境搭建，部署及运维成本高的问题，为微服务的大规模应用起到了推波助澜的作用。微服务管理是 PaaS 平台的核心，在 PaaS 平台对多系统、多应用、多服务的集中协同管理中，我们引入微服务架构来进行服务的治理。基于微服务的容器化 PaaS 平台应用管理架构如图 7-7 所示。

各系统业务应用以微服务的形式部署在容器云 PaaS 平台之上，由能力开放与微服务管控中心进行应用微服务统一管理，主要实现：

- 服务注册与发现。
- 服务编排。
- 服务升级与回退。
- 服务伸缩与漂移。
- 服务熔断、降级与隔离。

业务开发人员在基于微服务的容器化 PaaS 平台之上，通过服务拼装，可以快速实现业务能力上线，快速响应业务变革，不再需要等待漫长的 IT 改造时间。

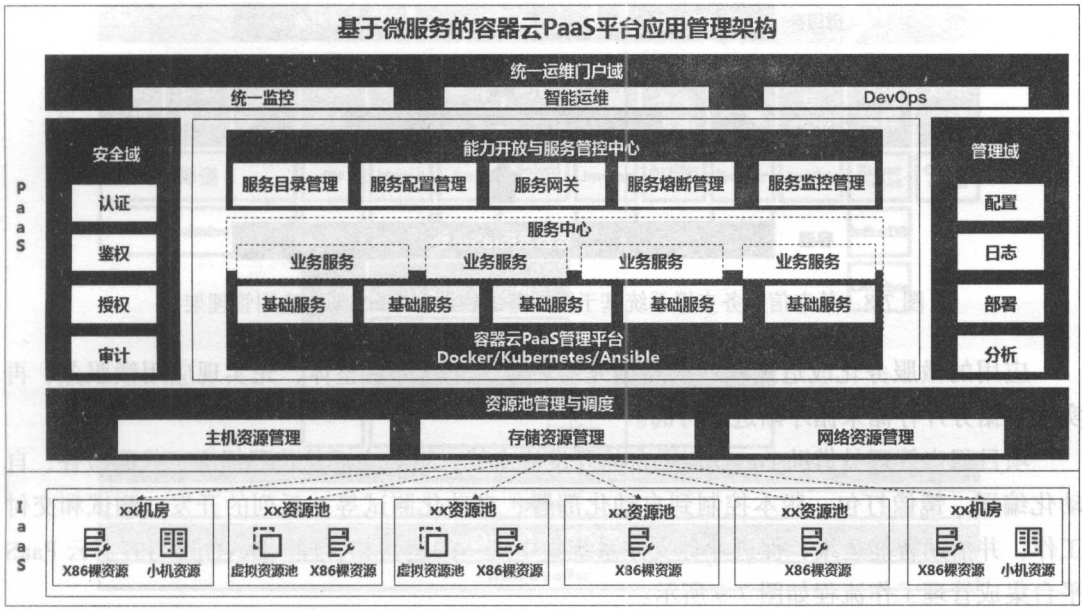


图 7-7　某电信业务支撑系统基于微服务的容器云 PaaS 平台应用管理架构

一个业务由多个微服务组织而成，服务之间会形成调用关系，调用链每一步的性能决定了业务的性能指标，因此，监控服务链各关键环节的性能指标也非常重要。容器云 PaaS 平台自动解析服务的“调用链”，通过收集、分析服务性能指标来实现对服务的综合监控。

只有采用微服务架构，才能使得 DevOps 的持续开发流水线发挥更大的优势。通过 DevOps 持续集成理念，支撑上层微服务的全生命周期管理，从根源上提速应用微服务的开发、变更、部署、测试和上线效率。

7.2.6　结合 DevOps 实现“云开发 + 云运维”的流水线管理

在前面的章节中我们已经详细介绍了 DevOps 的核心理念，在企业级容器云 PaaS 平台的运营实践中，将 PaaS 平台与 DevOps 模式更好地融合起来，发挥各自的优势和长处，以便实现复杂多种应用场景的持续开发、持续测试、快速部署和敏捷交付。容器、微服务和 DevOps 三者在应用实践中的关系可以通过图 7-8 来帮助我们理解。

容器云 PaaS 平台作为底层支撑，以 Docker 容器技术为主，也可以像 VMware 或 OpenStack 这样以 VM 为管理单元的方式，旨在为上层提供有 SLA（Service-Level Agreement）能力的资源池管理与调度。

DevOps 作为一层可选平台，以流程自动化、工具自动化为主要手段，为最终业务交付提供更敏捷、更数字化的能力。

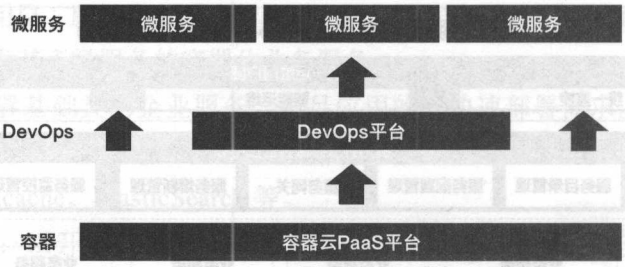


图 7-8 某电信业务支撑系统基于微服务的容器云 PaaS 平台应用管理架构

应用的微服务化改造过程一般遵循先易后难，先局部后整体，先实现应用微服务、再实现数据分片存储来循序渐进进行的。

项目租户管理员借助 DevOps 平台的开发流水线，可以实现从代码提交、代码检查、自动化编译、镜像打包、版本控制到自动化部署、自动化测试等一系列的开发、测试和交付工作，并借助智能运维，促进业务支撑系统集中统一的运营和管理。DevOps 与容器云 PaaS 平台集成管理工作流程如图 7-9 所示。

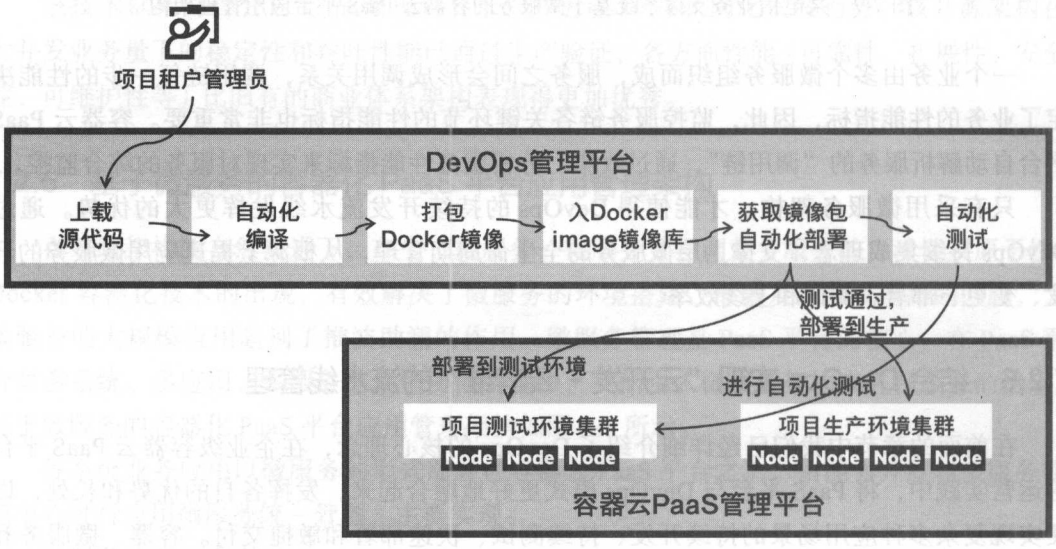


图 7-9 DevOps 与容器云 PaaS 平台集成管理工作流程

DevOps 平台集成了各种开源组件实现 CI/CT/CD（持续集成、持续测试、持续交付）的全过程管理。整个过程可以依托平台自动化工具一键完成，大大提升了应用发布效率，并且在效率提升的同时保证质量，降低了业务风险。与可视化监控工具集成，提供了开发代码质量、版本发布、测试结果、缺陷修复、运维监控指标等整个开发过程的可视化监控和管理。DevOps 流水线集成如图 7-10 所示。

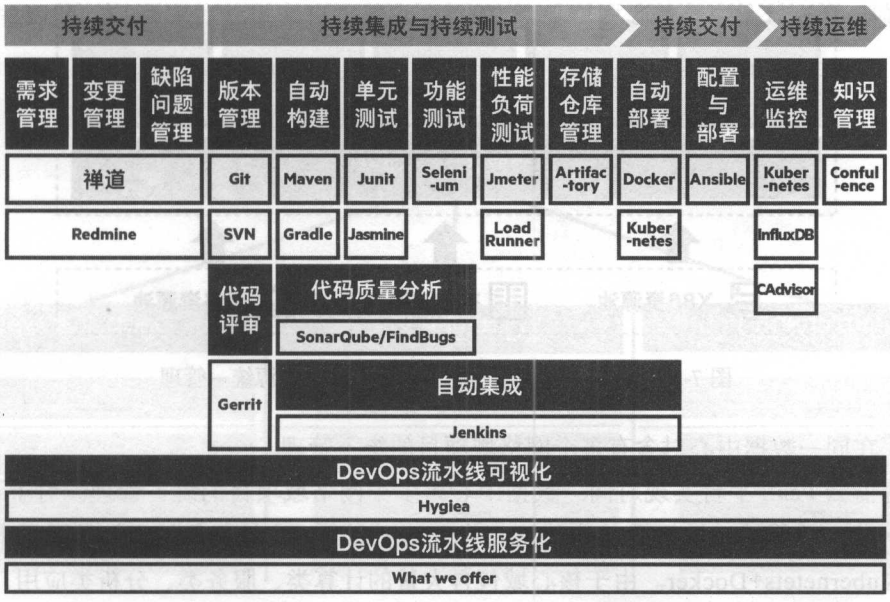


图 7-10　DevOps 流水线集成管理

7.2.7　容器云 PaaS 平台多集群管理方案

在企业级容器云 PaaS 平台的多集群管理方案中，我们一直在思考什么样的多集群管理模式最适合企业当前的业务支撑系统现状，是采用大一统的集中 PaaS 集群管理平台？还是采用松耦合的联邦式集群管理模式。PaaS 平台要承载多个大型项目，每个项目业务特点和技术构架都不一样，有大量主机，分布在多个机房和资源池，每个系统均跨多个网络域包含多个集群，地理分布跨全国 31 个省。既要保持各个项目的灵活性，构架不受冲击，又能实现集中式的资源和应用管理是 PaaS 多集群管理平台的设计原则。

因此我们最终采用了 Cluster Federation（集群联盟）的模式对各项目的多集群进行统一管理。采用此方式对各个项目的影响最小，系统不用做大的调整，同时可以实现资源集中管理和统一服务调度，以及统一的镜像管理、应用管理、配置管理、资源管理和监控管理等。

我们结合 Kubernetes 容器集群管理的优秀技术，通过自主研发的 Ku8 Manager 多集群管理软件，实现了多集群的统一管理，可以涵盖以下典型应用场景。

（1）在同一数据中心的多个项目共享资源统一管理

在容器云 PaaS 平台中为每个项目建立独立的租户，安全隔离不同租户的资源和访问权限。通过分区和分集群的方式，既保持各项目的独立性又保持资源的共享。为大的项目建立独立的 Clutser，小的项目通过分区实现安全隔离。多个项目共享资源统一管理如图 7-11 所示。

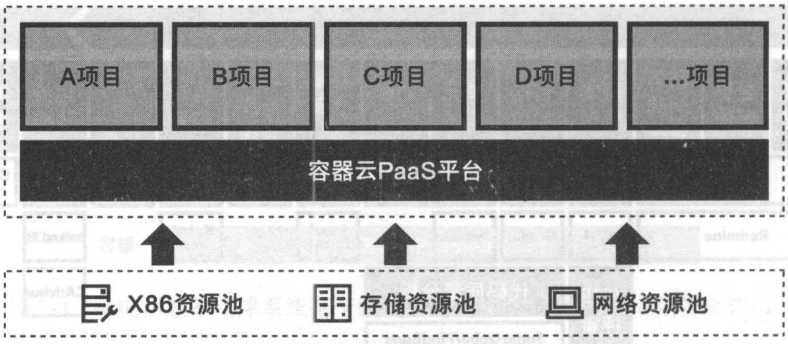


图 7-11 在同一数据中心的多个项目共享资源统一管理

（2）在同一数据中心对含有多个网络域项目的统一管理

由容器云 PaaS 平台实现对同一数据中心跨多个网络域项目的统一管理。对于 DMZ 域和互联网域，部署的一般都是 Web 类应用和 Proxy 应用，比较轻量但弹性扩展需求强，适合部署 Kubernetets+Docker。由于核心域包含大量的计算类、服务类、分析类应用，适合部署 Mesos+Kubernetes+Docker。跨多网络域的多集群统一管理如图 7-12 所示。

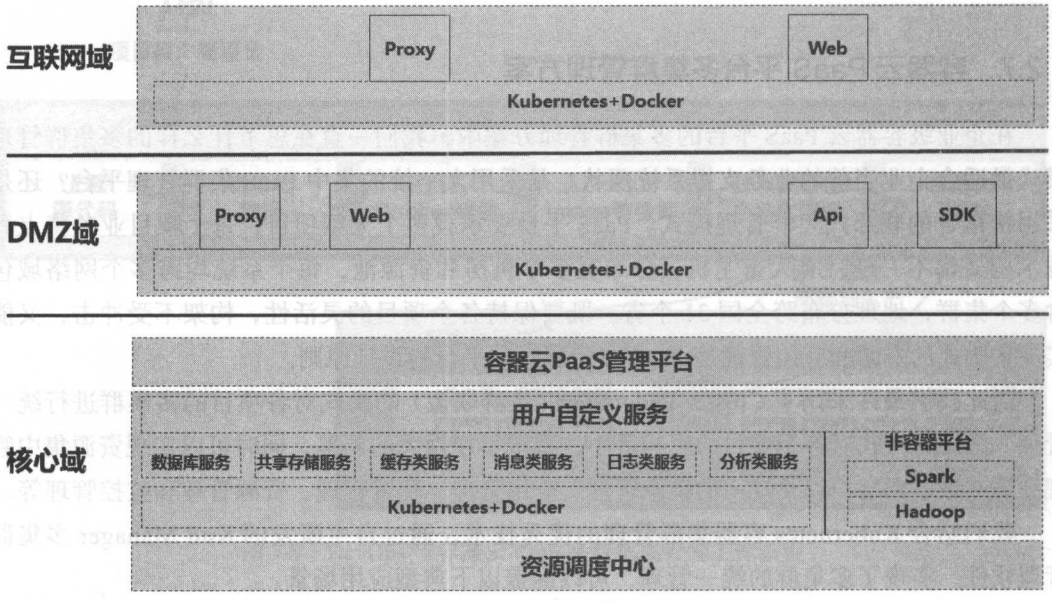


图 7-12 跨多网络域的多集群统一管理

（3）跨 IP 承载网的多数据中心统一管理

该电信业务支撑系统部署跨多个数据中心，多个数据中心分布在不同的地理区域，容器云 PaaS 平台可以对跨 IP 承载网的多数据中心实现统一的应用镜像管理、统一应用部署

和服务的调度，以及应用的快速容灾调度。跨 IP 承载网的多数据中心统一管理如图 7-13 所示。

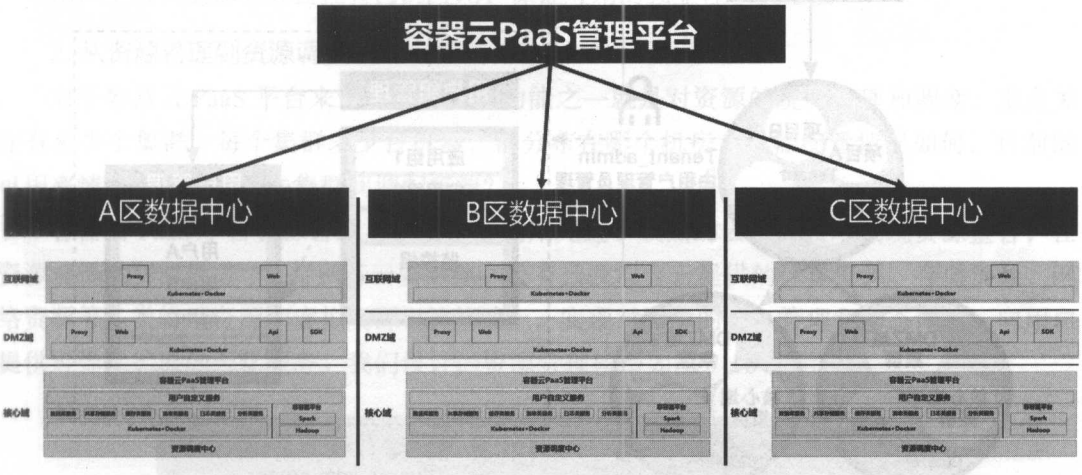


图 7-13　跨 IP 承载网的多数据中心统一管理

7.2.8　容器云 PaaS 平台建设中应关注的重点和难点

对于企业级容器云 PaaS 平台，在多集群管理的模式下，每一个功能都会变得非常复杂。我们对于建设中的重点和难点，从以下多个方面进行了详细的方案分析和设计实现。主要包含安全集中控制、资源统一管理、镜像统一管理、应用统一管理、配置统一管理、统一的监控管理和智能运维，以及企业基础服务的提供。

1. 安全治理

联合多集群模式下安全管控是需要重点考虑的一个非常复杂的因素，包括：

- 如何设计平台管理员、集群管理员、普通用户等多个角色的管控范围和权限分配？
- 如何定义租户？如何控制租户和集群资源的关系？如何映射租户和项目的关系？
- 每一个租户对于自己的集群资源如何进一步划分分区，分配权限给开发、部署和运维人员？
- 每个用户如何管理自己的应用和监控应用的健康情况？

考虑以上问题，我们在容器云 PaaS 平台中，通过租户、集群、用户三级安全管理，实现多集群管理的安全集中管控。安全管控模型如图 7-14 所示。

首先要控制对集群的安全访问，每个集群都采用安全模式启动，使用 CA 证书的安全认证方式对集群进行 SSL 安全访问。没有证书不允许访问任何一个集群。集群安全访问如图 7-15 所示。

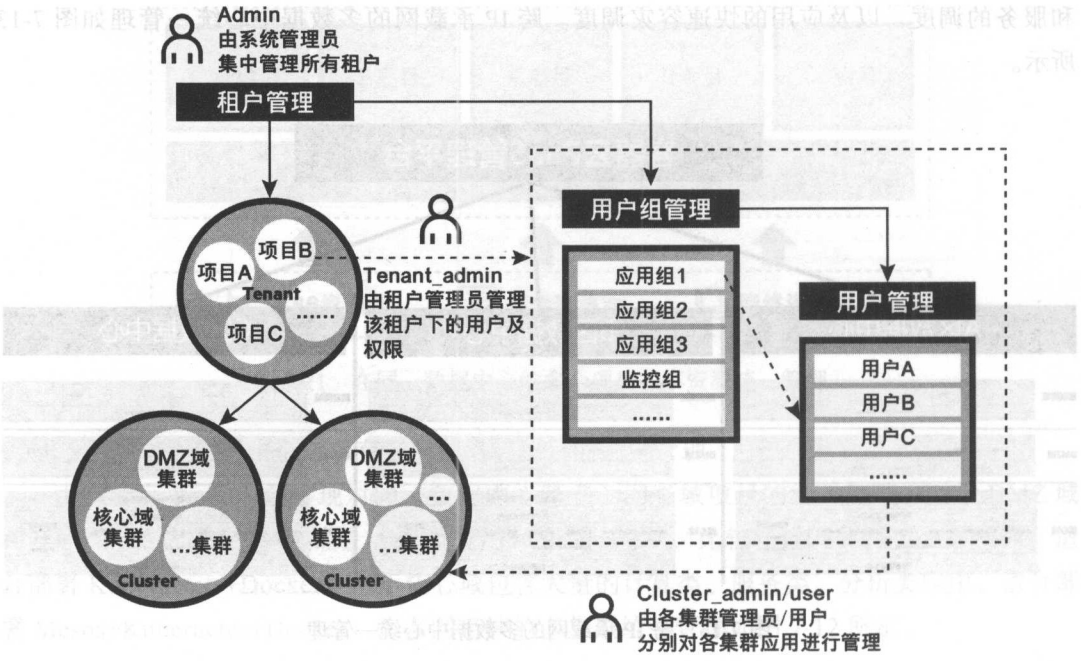


图 7-14 容器云 PaaS 平台的安全管控模型

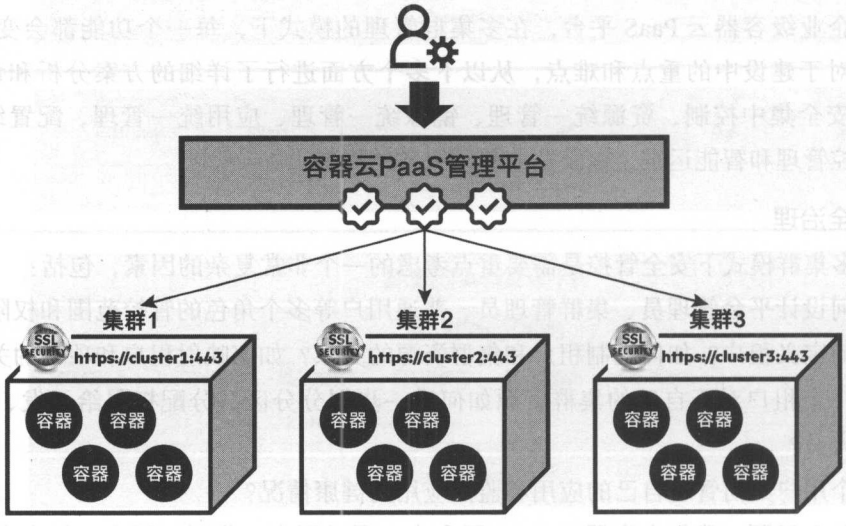


图 7-15 容器云 PaaS 平台的集群安全访问

在多集群管理中，租户为一个虚拟的单位概念，每个项目可定义为一个租户，如项目 A 即为一个租户，每个租户下可建立多个集群，租户拥有一个或者多个集群的访问证书，租户实现对自己名下资源的管理和分配。

租户再通过用户组绑定不同集群和分区资源，属于某用户组下的用户即可获得对资源的访问权限。

用户本身只能部署和监控自己的应用，保证应用的独立性和安全性。

2. 从资源管理到资源调度

对于容器云 PaaS 平台来说，最主要的功能之一就是对资源的统一管理和调度，重点关注有多少个集群，每个集群多少台机器，都分布在哪个机房？机器配置情况如何，目前的可用率情况如何，如何去集群间调配资源？

容器云 PaaS 平台资源管理与调度中心利用先进、成熟的云计算技术实现资源整合，在资源池提供资源供应（支持私有云资源池、公有云及混合云提供的计算资源、存储资源、网络资源等各类物理资源和虚拟资源）的基础上，实现对资源的统一管理和动态调配。向用户提供可弹性扩展的云化服务。我们设计的资源管理模型如图 7-16 所示。

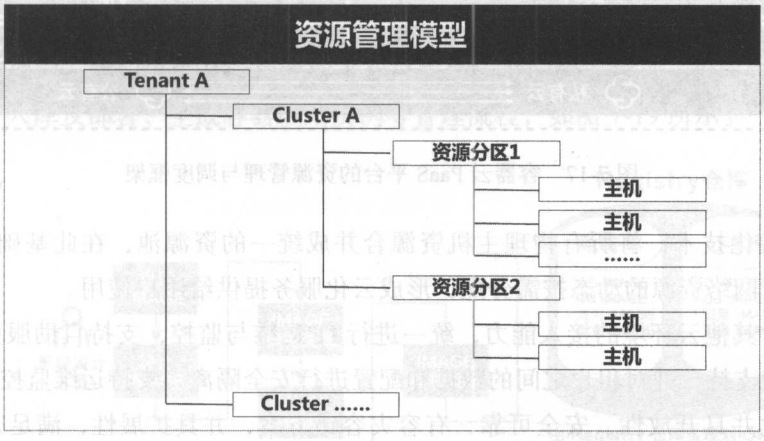


图 7-16　容器云 PaaS 平台的资源管理模型

基于以上资源模型，资源管理的流程如下：

- 经过证书的安全认证之后，通过 Kubernetes API Server 对集群进行纳管，把集群的 Node 的数量、配置等信息导入，同时补充机房、机架等信息。
- 将纳管集群和租户进行绑定，把集群分配给指定租户。
- 租户可以继续对集群资源进行分区，比如分成 WEB 区、DB 区等，便于部署不同的应用，并且可以实现不同配置的主机的共存。

容器云 PaaS 平台与资源层对接，将物理资源映射为逻辑资源，通过对逻辑资源的分区管理，实现租户间的资源隔离。同时，当项目集群内资源无法满足应用扩容需求时，可快速创建相应的资源模板，实现应用的快速动态扩容。资源管理与调度框架如图 7-17 所示。

容器云 PaaS 平台资源管理与调度中心具备如下能力：

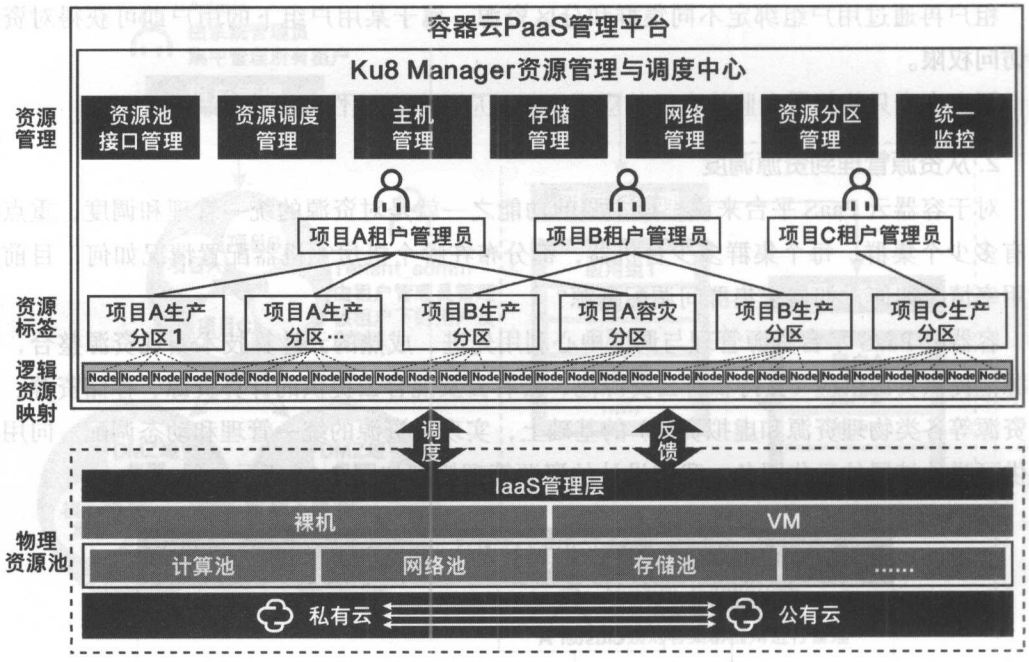


图 7-17 容器云 PaaS 平台的资源管理与调度框架

- 通过云化技术，将所有物理主机资源合并成统一的资源池，在此基础上实现计算、存储、网络资源的动态按需分配，形成云化服务提供给用户使用。
- 具有对其他云环境的接入能力，统一进行 IT 运维与监控，支持自助服务，提供多租户环境支持，并对租户之间的数据和配置进行安全隔离，支持运维监控能力。
- 标准化并具开放性，安全可靠，有容灾容错方案，并具扩展性，满足未来业务负载的支撑能力，方便维护，并具有可视化自助服务管理能力。

3. 镜像的全生命周期管理

镜像的统一管理是多集群模式下一个非常重要的功能，容器云 PaaS 平台需要对镜像做集中的管控，但是又要考虑镜像部署的效率。因此，在 PaaS 平台中我们设计了两级镜像的构架，如图 7-18 所示。

在主镜像库实现镜像的全生命周期的管理，镜像的入库、出库和版本更新都在主镜像库中统一集中管理，并和 DevOps 整合在一起，进行应用镜像统一的开发和持续运营管理。

每个集群都有一个子库或者几个集群共享一个子库。子库可以设定同步规则，只同步一部分镜像到子库，因此，每个项目可以只同步自己的镜像到自己的集群中，提高集群应用部署的效率，并实现镜像的跨地域、跨 IP 承载网同步和更新，支持全量同步和增量同步。

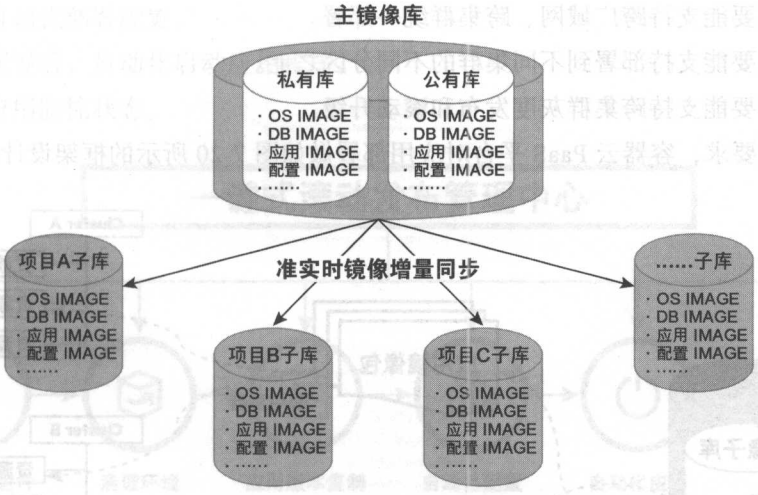


图 7-18　容器云 PaaS 平台的两级镜像管理架构

在镜像管理中，我们提供对应用镜像进行全生命周期的统一管理，提供镜像模板规划、设计、生成、入库及部署、生成容器应用实例等管理流程，如图 7-19 所示。

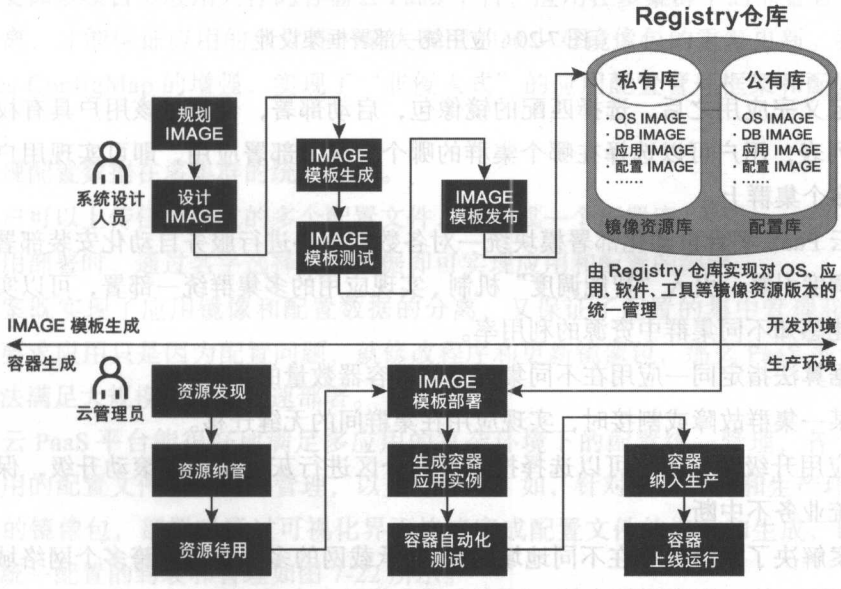


图 7-19　容器镜像进行全生命周期管理

4. 应用灰度发布和流水线部署

多集群情况下的应用部署也是容器云 PaaS 平台最主要的管理功能之一，需要考虑的情况会比较复杂：

- 应用需要能支持跨广域网、跨集群统一部署。
- 应用需要能支持部署到不同集群的不同分区之上。
- 应用需要能支持跨集群灰度发布和滚动升级。

针对这些要求，容器云 PaaS 平台对应用部署做如图 7-20 所示的框架设计。

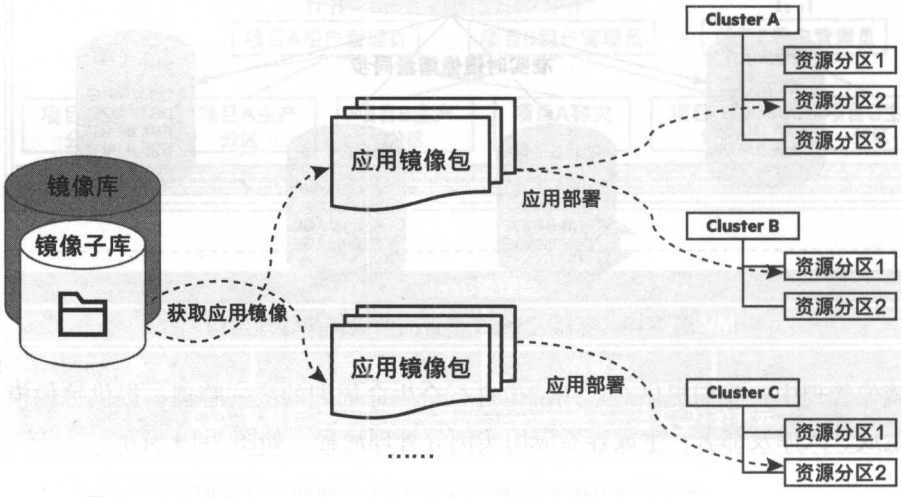


图 7-20 应用统一部署框架设计

用户定义完应用之后，选择匹配的镜像包，启动部署，会显示该用户具有权限的集群和分区的列表，用户可以选择在哪个集群的哪个分区上部署应用，即可实现用户一键式部署应用到多个集群上。

容器云 PaaS 平台自动化部署模块统一对各数据中心进行服务自动化安装部署。通过多集群统一调度引擎，引入“智能调度”机制，实现应用的多集群统一部署，可以实现：

- 智能感知不同集群中资源的利用率。
- 根据算法指定同一应用在不同集群上部署容器数量的比例。
- 在某一集群故障或割接时，实现应用在集群间的无缝迁移。
- 在应用升级时，用户可以选择按集群和分区进行灰度发布和滚动升级，保证上线时系统业务不中断。

该方案解决了系统分布在不同地域的跨 IP 承载网的多个集群、跨多个网络域智能的应用部署和调度。统一调度引擎支持一键式自动化滚动发布，如图 7-21 所示。

流程步骤如下：

- 停止当前正在运行的应用程序和相关联进程。
- 清理部署环境和配置项。
- 将应用版本复制到发布目录下。

- 进行自动化部署配置。
- 完成配置后，自动化启动应用程序。
- 进入应用监控状态。

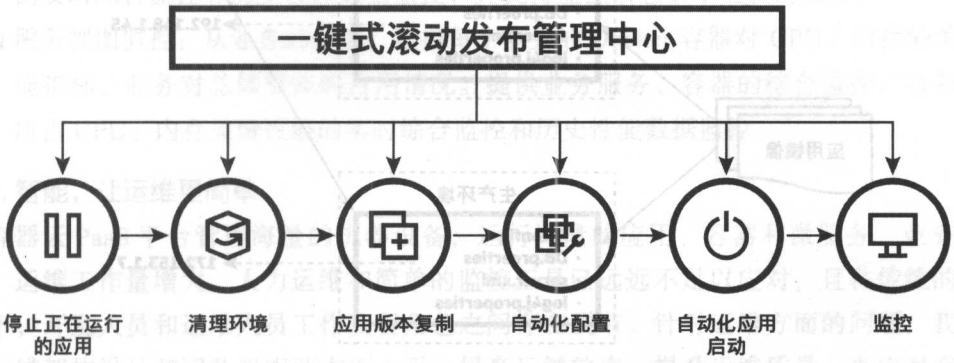


图 7-21　一键式自动化滚动发布流程

5. 配置分离，才能灵活管理

对于支撑多项目多应用共存的容器云 PaaS 平台，应用在多集群中的部署必须实现应用和配置分离，才能保证应用的独立性，最大限度的减少对镜像包的重复更新。我们通过对 Kubernetes ConfigMap 的增强，实现了“非侵入式”的应用配置管理框架和配置的统一管理。可以：

- 实现配置数据在多集群的统一定义。
- 用户可以上传任意格式的多个配置文件，绑定成一个配置库。
- 应用部署时，通过名字选择配置数据即可实现应用和配置的绑定。

该方案既实现了应用镜像和配置数据的分离，又保证了配置的集中管理和灵活更新。否则如果要求应用只是因为配置问题，就修改程序和更新镜像包，那么 PaaS 平台适应性则太差，无法满足大规模应用的快速部署。

容器云 PaaS 平台能很好地满足多应用的复杂环境下的配置统一管理，各个租户可以对私有应用的配置文件进行统一管理，以方便修改。如，针对测试环境和生产环境的区别，使用相同的镜像包，部署时通过可视化界面快速完成配置文件的修改和生成，即可完成部署工作。统一配置的封装和管理如图 7-22 所示。

用户可以提前上传或者直接拷贝多个任意格式的配置文件（XML、PROPERTIES、TXT 等）到不同的集群中创建配置库，形成一个绑定在不同的集群中的统一名称的配置库。

用户在应用部署时，选择对应名称的配置库，就可以实现应用在不同集群部署，自动绑定自己集群的配置数据，实现不同环境、不同集群的统一的应用和配置数据的绑定。

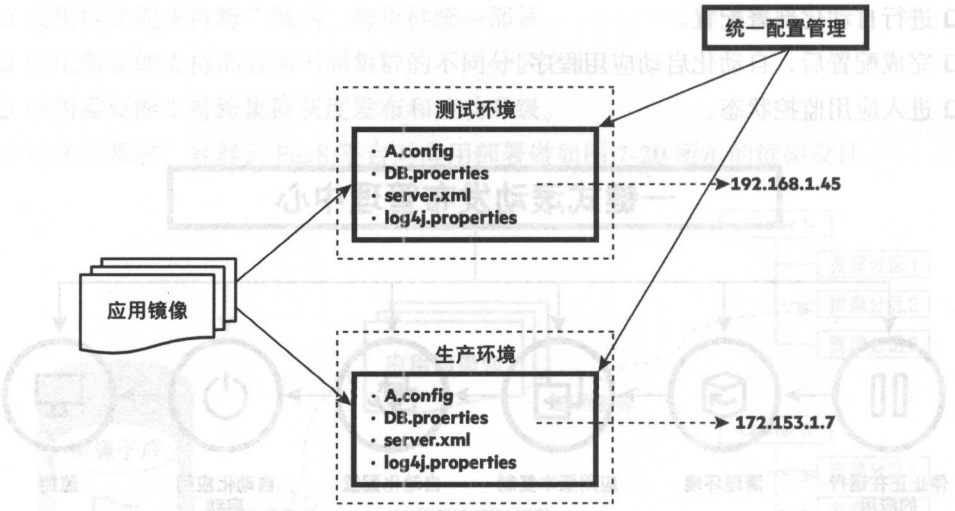


图 7-22 统一配置管理

6. 从服务视角的综合监控

在多集群多租户模式下，每个租户会关心自己的资源和服务的运行状况，因此容器云 PaaS 平台通过综合监控管理实现对所有租户、集群、资源、应用的集中数据采集、分析和展示，并能在发生故障时及时获取告警信息，快速进行故障隔离和恢复。

容器云 PaaS 平台能够提供对集群的综合监控、以主机视图的监控和以服务视图的监控。监控模型如图 7-23 所示。

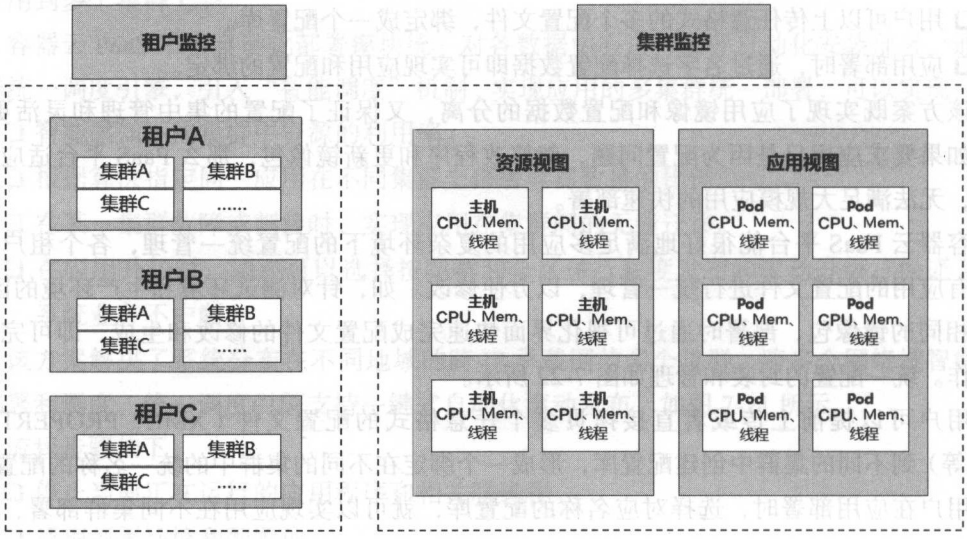


图 7-23 综合监控模型

- 集群综合监控：直观展示集群的综合监控信息，包含资源的使用情况、应用服务的资源使用情况、公共服务中间件对资源的占用情况、系统关键事件等信息。
- 主机视图监控：从设备的角度，采集主机 CPU、内存资源占用情况，提供设备资源的实时综合监控和历史性能数据监控，并支持监控信息展示更新的频率。
- 服务视图监控：从业务的角度，监控每个业务的 Docker 容器对 CPU、内存的关键性能指标，业务对总体资源的占用情况。提供业务服务、容器的综合监控，业务服务所占 CPU、内存关键性能的实时综合监控和历史性能数据监控。

7. 智能，让运维更简单

容器云 PaaS 平台管理海量的机器设备，运行海量的应用、容器和微服务，业务变更频繁，运维工作量增大，人力运维和简单的监控工具已远远不足以应对，且在传统的开发模式下，开发人员和运维人员工作流程角色之间存在脱节。针对运维方面的问题，我们从智能运维架构设计和团队组织两方面入手，提高运维效率，提升运维质量，来应对容器云 PaaS 平台中多租户、多项目集中统一的运维管理工作。

在架构设计方面，我们建立基于机器学习的智能运维框架，通过采集、建模、测量、分析、决策、控制来形成有效闭环不断改进的运维框架体系。提炼实际运维过程中的知识经验，设计自动化运维流程，建立多条自动化机器运维流水线，自动化运维流程参考图 7-24 所示。

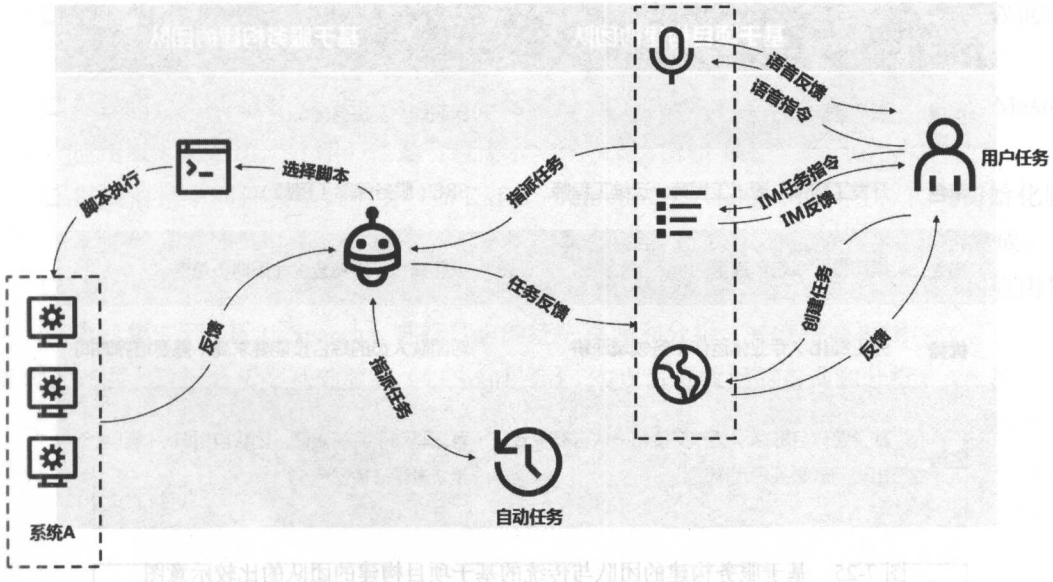


图 7-24　自动化运维流程管理

自动化的智能运维基于应用和系统日志，进行数据清洗、关联和标准化处理，借助机

器学习和智能分析算法，根据关键阈值进行预警，或自动优选匹配结果进行工单智能处理。主要包括：

- ❑ 自动设备巡检。
- ❑ 自动健康分析。
- ❑ 指标异常根源分析。
- ❑ 日常运维工作自动化。
- ❑ 自动处理工单，报表生成。
- ❑ 机器智能学习分析。
- ❑ 故障根源分析，影响分析。
- ❑ 智能推送经验知识。
- ❑ 运维助手机器人。
- ❑ 自动生成、发送检查报告。

在团队组织方面，改变传统的运维组织与管理模式，引入服务保障工程师（Service Reliability Engineer，SRE）角色，重新定义微服务团队的管理模型，运维团队从“基于项目团队”向“基于服务团队”转型，建立新型的SRE团队，从根本上解决开发和运维之间的矛盾，提升运维团队核心能力。基于服务构建的团队与传统的基于项目构建的团队的对比，如图7-25所示。

	基于项目构建的团队	基于服务构建的团队
类型	职能团队	跨职能产品服务团队
角色	开发工程师、测试工程师、运维工程师..	SRE（服务保障工程师）
模式	横切模式：各负其责	纵切模式：服务全生命周期负总责
优势	分工细化，专业化运作，各领域深耕	对团队人员的综合技能要求高，需要培养时间
劣势	缺少整体目标感，工作交接和确认容易脱节，出现问题必须界定清楚	目标驱动，共享责任，团队自组织，端到端负责，减少交接损耗

图7-25 基于服务构建的团队与传统的基于项目构建的团队的比较示意图

借助智能运维，在以下方面能够帮助企业获得收益。

- ❑ 提高运维效率：代替运维人员和系统人员的日常工作，运维自动化，提高人工运维

效率，降低人工成本。

- 提升运维质量：运维智能化，提升生产运维质量，高效解决业务运行过程中不断出现和可能出现的问题。
- 自动建议决策：自动巡检，自动执行任务，智能机器分析主动给出建议和辅助决策。
- 任务流程驱动：运维工作流程驱动，规范可视，机器智能为主，人力为辅，运营过程尽可能减少人为因素介入。
- SRE 专家保障：运维角色转型，以平台为基础，以智能为工具，配合服务稳定专家，保障 PaaS 平台可靠运行和服务的稳定性。
- 优化知识积累：通过机器深度学习自动积累并固化经验和知识，投入更少成本保障业务更高的质量（可用、安全、容量）。

8. 企业基础服务的提供

容器云 PaaS 平台作为基础技术设施平台，应预置各种常用开源的、稳定的基础类服务，以满足用户应用的快速部署需求。基础类的软件大部分是集群类的软件，容器化难度较高，要满足业务的高可用和高可靠要求，需要做很多的技术改造和性能提升。如图 7-26 所示，基础类的软件包括

- 提供数据库类基础服务（如 MySQL、PostgreSQL、MongoDB 等）：实现初始化脚本挂载、数据服务高可用和数据备份高可用等功能。
- 提供共享存储类基础服务（如 GlusterFS、Ceph 等）：实现初始化脚本挂载、数据高可用和支持 Volume 的定义等功能。
- 提供缓存类基础服务（如 Redis、Memcache 等）：实现初始化脚本挂载，保证 Master 应用服务高可用，并能够根据容量需求实现快速水平扩展。
- 提供消息中间件类基础服务（如 Kafka、RabbitMQ、ActiveMQ 等）：实现初始化脚本挂载，支持高可用，单点故障不会影响服务的正常使用，支持管理工具的集成。
- 提供统一日志采集和查询类基础服务（如 ElasticSearch 等）：采集系统日志和应用日志，集成到 ElasticSearch，进行日志的统一存储和分析，并支持高可用。
- 提供大数据分析类基础服务（如 Spark 等），实现海量数据的查询和分析。

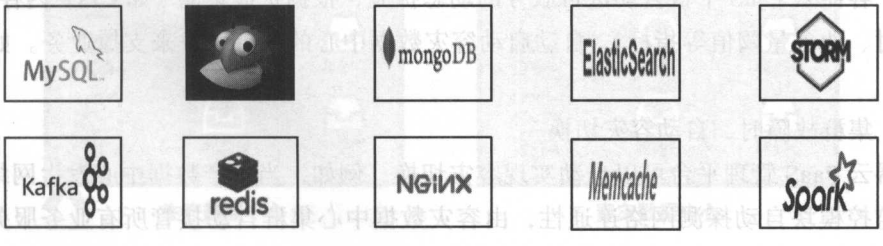


图 7-26　企业常用的基础类服务软件

9. 打造高可用的数据中心

容器云 PaaS 平台除具有良好的管理能力与水平可扩展能力外，还必须具备可靠的稳定性，应能够提供故障恢复、高可用性、灰度升级、弹性支撑等业务连续性保障措施，关键场景包括：

（1）多集群的统一服务部署和灰度升级

容器云 PaaS 平台自动化部署模块统一对各数据中心进行服务自动化安装部署。可以定义同一个服务在不同数据中心的 Kubernetes 集群统一部署，并且可以定义在每个集群部署服务的容器实例的比例。比如按 6:4 的比例在集群 A 和集群 B 上部署服务。

在进行自动化服务升级时，通用的做法是先在一部分集群部署新版本，稳定之后再平滑升级全部的节点，以实现服务不中断的灰度发布与版本升级。如图 7-27 所示。

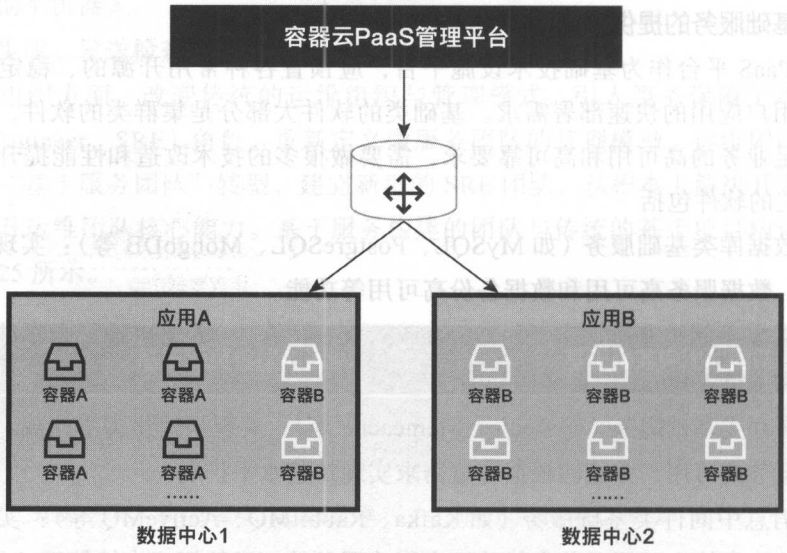


图 7-27　多集群的统一服务部署和灰度升级

（2）业务高峰弹性支撑

当生产数据中心的应用主机高负载运行产生告警，或业务高峰期当一个数据中心容量不足时，容器云 PaaS 平台自动进行服务的动态扩展，根据扩展策略（如 CPU/ 内存使用率、交易延时、业务量阈值等指标），自动启动容灾数据中心的部分服务来支撑业务。如图 7-28 所示。

（3）集群故障时，自动容灾切换

容器云 PaaS 管理平台可以自动实现容灾切换，例如，当生产数据中心发生网络整体故障时，监控模块自动探测网络连通性，由容灾数据中心集群自动接管所有业务服务，故障恢复后，切换到容灾集群的容器服务再根据优先策略进行恢复切回。如图 7-29 所示。

（4）容器故障时，自动重启或重建服务，保障应用高可用

容器云 PaaS 平台能够自动进行应用的健康检查，当生产数据中心集群内的应用容器运行故障时，系统自动重启或重建容器，以保证运行容器支撑业务的总能力不变。如图 7-30 所示。

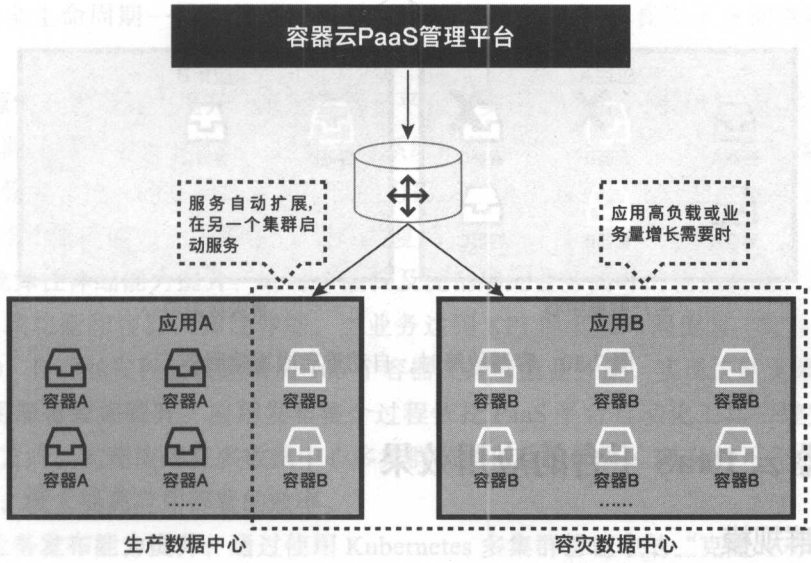


图 7-28　业务高峰弹性支撑

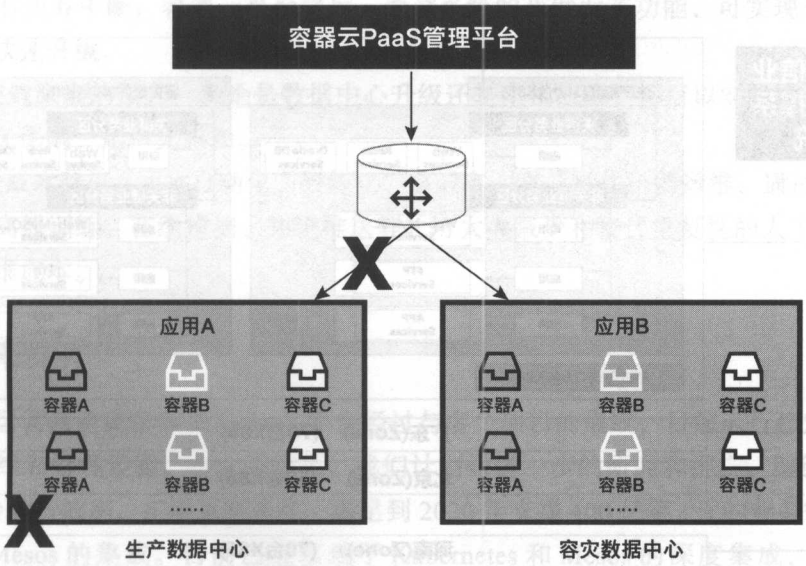


图 7-29　集群故障时，自动容灾切换

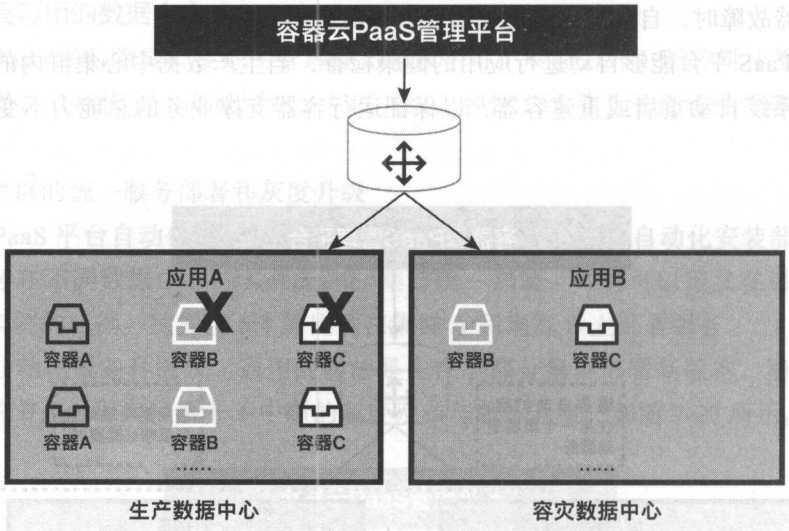


图 7-30 容器故障时，自动重启或重建服务

7.3 容器云 PaaS 平台的应用效果

7.3.1 集群规模

某电信运营商企业级容器云 PaaS 平台共管理了 33 个集群，由 1 200 多台 X86 组成，运行约 10 000 个容器，系统分布在全国。应用部署架构如图 7-31 所示。

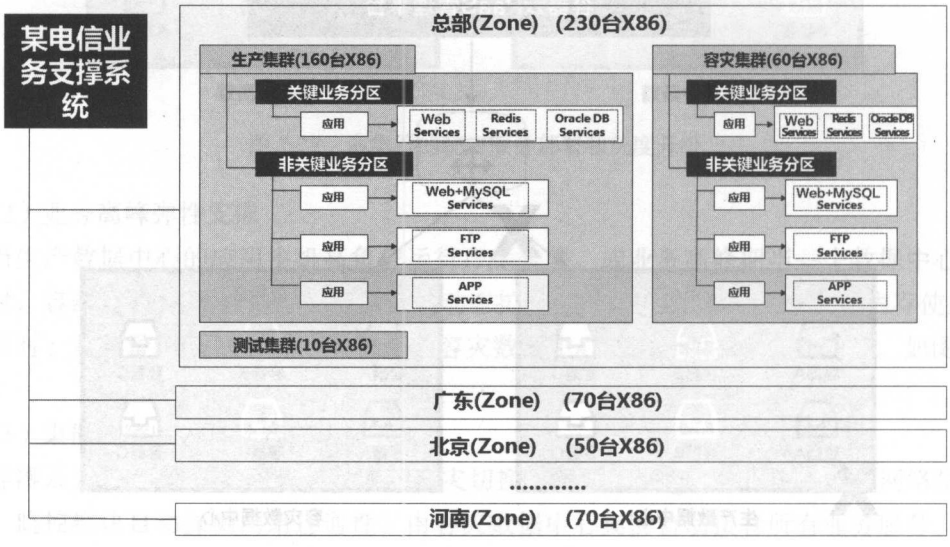


图 7-31 某电信业务支撑系统容器云 PaaS 平台部署架构

7.3.2　应用效果

某电信运营商企业级容器云 PaaS 平台从 2015 年开始云化改造，进行业务微服务化，引入了 Docker 容器，Kubernetes 集群管理和 MySQL、Tomcat、Kafka、Zookerper、Redis 等一整套的开源云化架构体系，并结合 DevOps 和智能运维，实现开发、测试到系统运维、软件交付的全生命周期一体化管理。通过不懈的实践和努力，在以下方面取得了较好的效果。

1）**资源统一管理能力提升**：通过容器云平台，实现了跨多个网络域、跨多个数据中心的复杂多集群环境下的集中管理，提高了对系统的管理和监控力度。统一的资源池化共享、统一的流程管理、统一的基础技术和应用能力提供、集中的平台化管控，提高了 IT 标准化水平，提升集中管控能力，并实现 IT 资产的复用。

2）**系统弹性伸缩能力提升**：针对节假日及促销期的业务高峰的剧烈波动，对业务节点实现动态负载均衡和容器的弹性伸缩，当业务达到 KPI 伸缩策略阈值时（交易量可达到 6 万笔 / 分钟），自动触发秒级快速增加 100 个容器支撑峰值业务量，实现快速支撑业务波动。

3）**应用部署效率提升**：应用发布整个过程依托 PaaS 平台自动化工具一键完成，采用流程驱动的方式，实现应用在多数据中心多集群间的自动化部署，10 分钟可完成 33 个集群的应用部署，极大提高应用部署的效率。

4）**新业务发布能力提升**：通过使用 Kubernetes 多集群管理中的“克隆”功能、能够秒级实现服务的发布。极大提高大规模应用快速部署的灵活性和系统快捷的水平扩展能力。

5）**应用升级的无业务中断率**：容器云平台上线之前，应用版本升级需要申请割接时间段，期间所有业务中断；容器云平台之后，通过系统的灰度发布功能，可实现全网无业务中断的应用快速升级。

6）**系统健康度的提升**：无论是数据中心升级还是主机故障，都可以实现业务自动容灾切换，业务无影响，用户无感知。

7）**运维成本降低**：通过自动化、智能化、高效率、高质量的机器运维，通过自动巡检、机器学习、离线训练、在线检测、知识推送等。可大大减少和替代重复性的人工运维工作，极大降低运维成本。

7.3.3　未来发展

某电信运营商企业级容器云 PaaS 平台经过与多个项目的集成，目前运行稳定，并积累了大量的运维和管理经验，在此基础上，我们计划做进一步的研究和推广。继续优化容器云 PaaS 平台运行效率，扩展系统规模，满足到 2020 年支撑 400 亿笔 / 天的业务需求。

1）与 Mesos 的集成。目前已经实现了 Kubernetes 和 Mesos 的深度集成，Kubernetes 可以作为一个 Framework 运行在 Mesos 之上。下一步准备迁移 Hadoop/Spark 到 Mesos，并

且实现 Kubernetes 和 Hadoop/Spark 应用的资源共享。

2）与 DevOps 工具集的集成。目前我们正在实现针对 PaaS 平台的 DevOps 理念的实现。包含工具、流程和监控。未来实现整个从开发代码质量分析→打包镜像→测试→部署的自动化支撑，并且通过 DashBorad 展现每个项目、每个微服务的所有环节的状态，从而提高 PaaS 平台对于整个开发测试过程的管控力度。

3）全面运用智能运维。将运维人员从繁琐的工作中解放出来，提升运维工作的效率。运维工作从依靠人工决策，逐步转变为依靠机器决策。智能化的监控系统和分析系统、预案管理系统结合起来，做到事前预警、事中恢复和事后存档，保证业务 7×24 小时高效稳定运行，运维部门将由传统的 IT 成本中心更多地转向 IT 服务中心、价值输出中心和利润输出中心转变。

4）形成云平台标准化技术规范。建立一套符合管理思路、适应性强、易于应用、易于推广的云平台标准化框架和模型，形成容器化应用的开发设计规范、镜像管理规范、PaaS 平台运维规范等，为未来的推广提供标准化和规范化支持，为企业 IT 系统的全面云化奠定技术基础。

作者简介

闫健勇 拥有超过15年的电信行业系统建设经验，主导了多项电信大型系统的架构设计和管理，对于云计算和大数据在电信行业中的应用拥有丰富的经验。

屈晓萌 拥有18年电信行业从业经验，负责过多个大型电信运营支撑系统的管理及规划设计，精通企业架构的端到端体系规划、构建、实施和运营，为企业提供面向下一代数字化运营模式的企业架构设计和转型实施工作。

王健飞 拥有超过16年的IT系统实施和管理经验，在电信企业BSS IT支撑以及企业架构规划等领域具有丰富经验，主要专注于电信行业IT支撑系统咨询、实施和管理工作。

吴治辉 拥有超过15年的软件研发经验，专注于电信软件和云计算方面的软件研发，拥有丰富的大型项目架构设计经验，是业界少有的具备很强Coding能力的S级资深架构师。

龚 正 拥有十多年IT从业经验，具备丰富的云计算、大数据分析和大型企业级应用的架构设计和实施经验，是电信、金融、互联网等领域的资深专家。

王 伟 拥有多年的IT从业经验，参与过多个大型应用的架构设计、系统开发和实施落地，精通大数据、云计算及大型系统架构和开发的相关技术，是云计算和大数据方面的技术专家。

刘晓红 拥有15年电信行业大型IT支撑系统建设经验，精通企业IT系统咨询规划、架构设计、实施运营全过程管理工作，专注于云计算、大数据、能力开放、智能运维在电信行业应用的新技术研究和整体解决方案提供。

封面设计 锡 彬

Architecture Development Guide for Enterprise Level Container Cloud

全书分为五大部分。第一部分（第1章）对云计算进行了概要性介绍，使读者建立起对虚拟化、容器技术、公有云和私有云的基本概念；第二部分（第2章）对微服务架构的设计和实现进行了介绍；第三部分（第3章）关注研发生产力，介绍了DevOps的概念和实践；第四部分（第4章、第5章、第6章）对Docker、Kubernetes、Mesos分别进行了介绍；第五部分（第7章）介绍了企业级容器云在电信行业的应用实践，使读者能够从容器技术，到大规模容器集群管理架构，到不同分布式应用混合架构的一系列技术，再到企业级容器云的应用实践均能够有所了解。本书的五个部分既彼此独立，又相互关联，帮助读者建立起云计算和容器技术的完整技术储备。

上架指导：计算机/云计算/网络

ISBN 978-7-111-58748-4

9 787111 587484 >

定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn